Edition Notice

Note Before using this information and the product it supports, read the general information under Notices.

First Edition (May 1999)

This edition applies to OS/2 Warp Server for e-business and to all subsequent releases and modifications until otherwise indicated in new editions.

About this book

This book is a technical reference for application programmers creating OS/2 (R) control program (kernel) functions. It contains APIs for OS/2 Warp Server for e-business. This book is intended to be used in conjunction with other books containing APIs that apply to OS/2 Warp Server.

Who should read this book

This book is intended for application programmers who want to use kernel functions in their programs. This reference provides technical information about functions and data structures available to the developer.

Conventions and terminology used in this book

The following conventions are used in this book

- **Boldface type** indicates the name of an item you need to select, field names, parameters, and folder names. It also indicates controls (when used in procedures), for example
  - Menu bar choices
  - Radio buttons
  - Push buttons
  - List boxes
  - Check boxes
  - Entry fields
  - Read-only entry fields
- *Italic type* indicates new terms, book and diskette titles, or variable information that must be replaced by an actual value. It also indicates words of emphasis and technical terms when introduced.
- `Monospace type` indicates an example (such as how to enter a command), text that is displayed on the screen, text you type, or special characters.
- UPPERCASE TYPE indicates a file and directory name, command name, or acronym.

Prerequisite and related information

This reference is intended for application designers and programmers who are familiar with the following

- C Programming Language
- Information contained in the following books *Control Programming Guide* , *Presentation Manager* , *OS/2 LAN Programming Guide and Reference* , and *Physical Device Driver Reference* .

DosDebug Commands

This chapter contains an alphabetic list of the following debug commands.

Cmd No.	Command Name	Description
33	DBG_C_Attach	Attach to a Process Command
34	DBG_C_Detach	Detach from a Process under Debug Command
35	DBG_C_RegDebug	JIT (Just-in_Time) Debugger Registration/De-Registrati Command
36	DBG_C_QueryDebug	Query JIT (Just-in-Time) Debugger Registered Command

----

DBG_C_Attach

Debug Command 33 Debug Attach Command

Parameters

Addr	Possible values are shown in the list below

0x00000000	The default action is to sever the connection between the debugger and the program being debugged, if a system resource is being held.
0x00000001	The sever action is not wanted between the debugger and the program being debugged.

Pid	Process ID of debuggee
Tid	Reserved, must be zero
Cmd	DBG_C_Attach
Value	Debugging Level Number

The only permitted debugging level number is shown in the following list

1	= DBG_L_386

This must be the first DosDebug command called when dynamically attaching to a process. No other DosDebug command will be accepted until the debugging connection has been established except for DBG_C_RegDebug to register a JIT (Just-in-time) debugger on a per-process basis and DBG_C_QueryDebug to query the JIT debug information. See DBG_C_RegDebug and DBG_C_QueryDebug for more information.

Returns

This command establishes a debugging connection. It must be the initial command, since it verifies the buffer format for the rest of the connection.

Because DosDebug usually cannot be ported to new machines without changing the format of the buffer, this command is needed to establish that the debugger is capable of handling the desired buffer format.

If the requested debugging level is not supported, an error is returned, and the connection is not made. This gives the debugger a chance to try again or to start automatically a different debugger process that uses a different buffer format.

For this command, the machine-independent and PID portion of the buffer is examined. This portion includes the Pid, Tid, Cmd, and Value fields. This makes it possible to port the DosDebug buffer from one machine to another without returning an error to the debugger on the initial DosDebug command.

The only DosDebug notifications that are returned by this command are DBG_N_Success and DBG_N_Error.

Restrictions

This DBG_C_Attach command does not require that the session for the program being debugged tohave been started with EXEC_TRACE, or SSF_TRACEOPT_XXX option by DosExecPgm or DosStartSession, as DBG_C_Connect requires.

If a connection to the program being debugged is established by a debugger, then another debug session cannot attach to the program being debugged while the first debugger is attached.

Remarks

If *Addr* is not 0, the connection between the debugger and the program being debugged is not severed. If any threads of the program being debugged, other than the thread that encountered the debug event, are holding system semaphores, they will be allowed to run until they release the semaphores. They will then be stopped, and the notification will be delivered.

If the thread encountering the debug event is holding a system semaphore, the connection between the debugger and the program being debugged is severed by terminating the program being debugged and returning a DBG_N_Error notification to the debugger with the value field set to 0 and the register set filled in. No further DosDebug commands will be accepted by the program being debugged, nor will it generate any other notifications.

If a DBG_C_Stop is issued, and a thread owning a system semaphore is about to generate a DBG_N_AsyncStop notification, it will be allowed to continue execution until it releases the semaphore. It will then be stopped, and the notification will be delivered. This is the only exception to the severing of the debugger/debugbee rule.

If *Addr* is set to 0, the connection between the debugger and the program being debugged is severed if a system resource is being held, in which case DosDebug returns

Tid	Thread owning semaphore
Cmd	DBG_N_Error
Value	ERROR_EXCL_SEM_ALREADY_OWNED

If the debugger needs to present some information to the user or use the thread holding the system resource, the debugger must terminate the program being debugged. Any other action might result in a system halt.

Upon attach to a process, a series of notifications will occur. The notifications include the current EXE module notification, thread (all that exist in the debuggee) create notifications, and currently loaded modules (DLLs) notifications. The notifications occur as DBG_NPModuleLoad, DBG_N_ThreadCreate, etc, just as they do with the DBG_C_Connect command.

DBG_C_Detach

Debug Command 34 Debug Detach Command

Parameters

Pid	Process of ID of debuggee
Cmd	DBG_C_Detach

Returns

This command detaches from the debugee connection. It is the last comand issued before resuming the process.

The only DosDebug notifications that are returned by this command are DBG_N_Success and DBG_N_Error.

Restrictions

Detach only works on a debuggee process currently under debug using the attach command, DBG_C_Attach. You cannot use DBG_C_Detach if you used DBG_C_Connect. DBG_C_Attach and DBG_C_Detach are paired and are used for debugging a process that is already running.

Remarks

By using this function, a debugger can only remove debug context of a given debuggee process as stated above. If the debugger needs to detach and have the debuggee terminate, it is neccessary to use DBG_C_Term command instead of DBG_C_Detach. This will terminate the debuggee process and remove attach information.

DBG_C_RegDebug

Debug Command 35 JIT (Just-in-Time) Debugger Registration/De-Registration Command

Parameters

Pid	Process of ID of debuggee
Cmd	DBG_C_RegDbg
Buffer	Pointer to JIT Debugger path name and arguments

Address of the buffer in which the fully-qualified path name of the JIT debugger is specified.   The path name can be followed by an optional arguments to the JIT debugger. If %d is found in the arguments, the system will replace %d with the ID of the failing process. If %d is not found in the arguments, the system will assume argument one is the process ID.

A coded example of this follows

Assume the ID of the failing process is 99.

If Buffer contains "C \OS2\MYJITDBG.EXE /Tn /K /P%d", the system will launch the JIT debugger as "C \OS2\MYJITDBG.EXE /Tn /K /P99"

If Buffer contains "C \OS2\MYJITDBG.EXE /Tn /K", the system will launch the JIT debugger as "C \OS2\MYJITDBG.EXE 99 /Tn /K"

Len 0 or the size of Buffer in bytes

A Len of 0 is used to deregister the JIT debugger from the given process. Buffer is ignored in this case. If Len is 0 and no JIT debugger is registered the system will return error code ERROR_INSUFFICIENT_BUFFER.

Addr Registration Flags

JIT_REG_INHERIT	Enable children processes to inherit registered per-process JIT debugger from parent. (This is the default.)
JIT_REG_NONINHERIT	Do not allow inheritance of per-process JIT debugger to the children of that parent process.

If Inheritance is being used with DosStartSession() on a PM application, the inheritance link is broken. This is due to the design of sessions management for DosStartSession() which causes all children processes to always inherit from the PM. The recommended way to start a child process is through DosExecPgm() under the same session type. The parent-child relationship will be set up correctly and the JIT will be inherited. Otherwise the parent application has to register the JIT on every DosStartSession() child process. The JIT could also be registered on the main PMSHELL.EXE process. This would cause all future DosStartSession() processes to inherit the JIT information from PM. This works around the DosStartSession() inheritance problem above.

Value Return error code, if any

Returns

The only DosDebug notifications that are returned by this command are DBG_N_Success and DBG_N_Error.

Valid Field return error return code(s)

ERROR_FILE_NOT_FOUND File was not found in specified path

ERROR_INSUFFICIENT BUFFER Returned if JIT Debugger not registered and a Len of 0 passed into register

Restrictions

This is one of the only DosDebug commands that can be called without having to issue a DBG_C_Connect or DBG_C_Attach first. This command is usually called after starting a process using DosExecPgm() or DosStartSession() in order to gather the PID to use with the registration command. Another way of gathering PID information is to use the DosQuerySysState(). This will return all of the current Process ID's running in the system. See DosQuerySysState() for more information. The registration of the debugger must use a fully-qualified path name and executable for per-process and global registration. Registration will not occur if the debugger is not physically found on the disk upon registration. If you make multiple calls to DosDebug with the DBG_C_RegDbg command, the previous debugger will be deleted and the newest one will be registered. If the DBG_C_RegDbg is called with the size field of 0 and a JIT debugger exists, the JIT debugger will be unregistered. This only applies for per-process JIT registration. Global registration cannot be unregistered. This is set for a systemwide level.

If a per-process JIT registration exists, it will be used over the global JIT registration specified in the CONFIG.SYS.

Remarks

The JIT debugger is invoked when an application process encounters a trap or exception not serviced by that application's exception management. Under normal operations, where a JIT debugger has not been registered with the OS, the application would be terminated by the user on a Hard Error Popup. This disallowed state information from being gathered. By registering a JIT debugger with the OS, the OS will launch the JIT debugger in place of the Hard Error popup.

The JIT debugger should attach to the dying process allowing the user to debug the dying process using a conventional debug program or just gather state debugger (unattended). A state debugger would gather required information to determine the state of the process dying; e.g., stack, registers, addresses, storage, etc. Once the state is gathered, the state debugger could save it to a log and terminate the offending process and/or start a new process in place of the old. This is completely customizable in the JIT debugger.

There are two types of JIT registration supported in OS/2. The first type is global registration. This allows the user to register a global debugger for the entire OS in the CONFIG.SYS. This debugger will be launched for any process in the OS that has an error. The global debugger will not support launching a PM (Presentation Manager(R)) JIT debugger. This has to be a VIO application (or one that produces no screen output) because of the organization of the boot cycle at which a JIT debugger can be invoked. The global JIT registration is done before the loading of device drivers, IFS, CALLS, and RUNS. This enables JIT support for all of these types of files.

The syntax for the CONFIG.SYS is as follows

```
JITDBGREG=[JIT_PathName] [arguments]
```

Refer to Buffer under parameters section above to see how [JIT_PathName] and [arguments] are used.

Example

```
JITDBGREG=c \os2\MYJITDBG.EXE /Tn /K /PID%d
```

The second type of JIT registration is the per-process registration. This type of registration allows the user to register any type of debugger including PM and VIO using the DBG_C_RegDbg command above.

Attention The JIT debugger writer needs to be aware of the environment that the JIT is being used in because of PM and VIO considerations for starting the debugger in regards session management and screen groups.

Note: For kernel debugger users there is an option to turn off the JIT debugger support when trying to catch traps in the kernel debugger. See the .on, .of and .oq switches in the kernel debugger help.

DBG_C_QueryDebug

Debug Command 36 Query JIT (Just-in-Time) Debugger Registered Command

Parameters

Pid	Process of ID of debuggee (Required for per-process entry)	
Cmd	DBG_C_QueryDebug	
Addr	Registration Flags	
	DBGQ_JIT_GLOBAL	Query registered global debugger
	DBGQ_JIT_PERPROC	Query registered per-process debugger
Buffer	A pointer to the buffer where the fully-qualified path name of the JIT debugger is returned.	
Len	Size of Buffer in bytes	
Value	Error code, if any	
	ERROR_INSUFFICIENT_BUFFER Buffer too small	
	ERROR_INVALID_FLAG_NUMBER Invalid Registration Flag	

Returns

This command returns the specified query form the operating system of the registered debugger. The buffer will contain NULL if no debugger is registered.

The only DosDebug notifications that are returned by this command are DBG_N_Success and DBG_N_Error.

Remarks

The DBG_C_QueryDbg command returns the registered JIT debugger from a process or the system (Global). The buffer contains the JIT debugger full path name and any arguments specified to the debugger.

Device Helper (DevHlp) Services and Function Codes

DevHlp services include

Service	Code	Description
DevHlp_CloseFile	80h	Close file (system initialization time only)
DevHlp_FreeCtxHook	64h	Free context hook
DevHlp_FileOpen	7Fh	Open file (at initialization)
DevHlp_GetDosVar	24h	Return address of kernel variable
DevHlp_KillProc	7Dh	Kill process unconditionally
DevHlp_OpenFile	7Fh	Open file (system initialization time only)
DevHlp_PerfSysTrace	45h	Write Software Trace information to STRACE buffer
DevHlp_QSysState	7Eh	Get system status information
DevHlp_ReadFile	81h	Read (system initialization time only)
DevHlp_ReadFileAt	82h	Seek and read (system initialization time only)
DevHlp_RegisterKDD	83h	Register driver with kernel debugger
DevHlp_SysTrace	28h	Add information to System Trace buffer

System event notification

Event	Index	Description
event_POWER	9	Power off event

DevHlp_CloseFile

DevHlp_CloseFile is used by base device drivers to close a file previously opened using DevHlp_OpenFile.

Calling Sequence in Assembler

```
LES   DI, FileClose
MOV   DL, DevHlp_CloseFile

CALL  [Device_Help]
```

ES DI points to a FILEIOINFO structure defined as follows

```
FILEIOINFO struc
length        dw     2    ; length of imbedded fle system operation structure
;                           must contain value 2 for CloseFile
FCLOSE struc
reserved      dw     ?    ; reserved
FCLOSE        ends
;
FILEIOINFO    ends
```

Results in Assembler

C Clear if the file is closed. AX = zero..

C Set if error. AX = Error Code. Possible errors

24 **ERROR_BAD_LENGTH**	The length in the FILEIOINFO structure is invalid.

Calling Sequence in C

```
#include  "dhcalls.h"
```

USHORT APIENTRY DevHlp_CloseFile (PFILEIOINFO pFileClose)

pFILEIOINFO   input   Pointer to the FILEIOINFO structure defined as follows

```
typedef struct FOPEN {
        PSZ    FileName;   /* (input) pointer to file name */
        ULONG  FileSize;   /* (output) size of file returned by FileOF
} FILEOPEN;

typedef struct FCLOSE {
        USHORT  reserved   /* reserved */
} FILECLOSE;

typedef struct FREAD {
        PBYTE Buffer;      /* (input) pointer to input buffer */
        ULONG ReadSize;    /* (input) number of bytes to read fromfile
} FILEREAD;

typedef struct FREADAT {
        PBYTE Buffer;        /* (input) pointer to input buffer */
        ULONG ReadSize;      /* (input) number of bytes to read from f
        ULONG StartPosition  /* (input) starting file position relativ
                                 the beginning of the file */
} FILEREADAT;

typedef union FILEIOOP {
        struct FOPEN FileOpen;
        struct FCLOSE FileClose;
        struct FREAD FileRead;
        struct FREADAT FileReadAt;
} FILEIOOP;

typedef struc _DDFileIo {
        USHORT Length; /* (input) length of imbedded structure */
        FILEIOOP Data; /* (input) imbedded file system operation struc
} FILEIOINFO, FAR * PFILEIOINFO
```

Results in C

Success Indicator 0 if file was closed..

24 **ERROR_BAD_LENGTH**	Length in the FILEIOINFO structure is invalid.

Remarks

DevHlp_FileClose may be called at initialization time only. It provides a primitive interface to the mini-IFS or micro_IFS at initialization.

Using this interface, one file only may be opened at a time. No handle is assigned by open. The file closed is assumed to be the most recent opened using DevHlp_OpenFile.

DevHlp_FreeCtxHook

DevHlp_FreeCtxHook frees a context hook allocated by the DevHlp_AllocateCtxHook.

Calling Sequence in Assembler

```
MOV   EAX, Hook_Handle
MOV   DL,  DevHlpFreeCtxHook
CALL  DeviceHelp
```

Results in Assembler

C	C Clear if hook freed.
	EAX = 0
C	C Set if error.
	EAX = Error code

Calling Sequence in C

```
#include  "dhcalls.h"
```

USHORT APIENTRY DevHelp_FreeCtxHook (ULONG HookHandle)

Results in C

Success Indicator 0 if hook successfully freed.

Remarks

The state of the interrupt flag is not preserved across calls to this DevHlp.

DevHlp_GetDosVar

DevHlp_GetDosVar returns the address of a kernel variable.

Calling Sequence in Assembler

```
MOV   AL, index               ; Index wanted.
MOV   CX, VarMember           ; Only used by index 14 and 16.
MOV   DL, DevHlpGetDOSVar

CALL  DeviceHelp
```

Results in Assembler

C Clear if successful. AX BX points to the index.

C Set if error.

Calling Sequence in C

```
#include  "dhcalls.h"
```

USHORT APIENTRY DevHelp_GetDOSVar (USHORT VarNumber, USHORT VarMember, PPVOID KernelVar)

VarNumber (USHORT)

The index into the list of read only variables

```
DHGETDOSV_SYSINFOSEG                                    1
DHGETDOSV_LOCINFOSEG                                    2
DHGETDOSV_VECTORSDF                                     4
DHGETDOSV_VECTORREBOOT                                  5
```

DHGETDOSV_VECTORMSATS	6
DHGETDOSV_YIELDFLAG	7
DHGETDOSV_TCYIELDFLAG	8
DHGETDOSV_DOSCODEPAGE	11
DHGETDOSV_INTERRUPTLEV	13
DHGETDOSV_DEVICECLASSTABLE	14
DHGETDOSV_DMQSSELECTOR	15
DHGETDOSV_APMINFO	16
DHGETDOSV_APM11INFO	17
DHGETDOSV_CPUMODE	18
DHGETDOSV_CPUMODE	19
DHGETDOSV_TOTALCPUS	20

VarMember (USHORT)

Applicable only to VarNumber 14 or 16.

For VarNumber = 14

VarMember=1	(Disk) has a maximum of 32 entries in the DCT.
VarMember=2	Mouse) has a maximum of 3 entries in the DCT.

For VarNumber = 16

VarMember=0	Query presence of APM BIOS.
VarMember=1	Query presence of APM BIOS and establish connection.

KernelVar (PPVOID)

Pointer to the address of requested variable to be returned.

Results in C

Success Indicator	Clear if successful; returns address of the requested variable in KernelVar.
Possible errors	None.

Remarks

The following table contains the list of *read-only* variables

Index	Variable Description
1	GlobalInfoSegWORD. Valid at task time and interrupt time, but not at INIT time. See below.
2	LocalInfoSegDWORD. Selector/segment address of local information segment for the current Local Descriptor Table (LDT). Valid only at task time. See below.
3	Reserved.
4	VectorSDFDWORD. Pointer to the stand-alone dump facility. Valid at task time and interrupt time.
5	VectorRebootDWORD. Pointer to restart the operating system. Valid at task time and interrupt time.
6	Reserved.
7	YieldFlagBYTE. Indicator for performing yields. Valid only at task time.
8	TCYieldFlagBYTE. Indicator for performing time-critical yields. Valid only at task time.
9	Reserved.
10	Reserved.
11	DOS session Code Page Tag pointer DWORD. Segmentoffset of the DOS sessions current code page tag. Valid only at task time.
12	Reserved.
13	Interrupt Level
14	DeviceClass Table (See DH_RegisterDeviceClass)
15	DMQS Selector Point to XGA adapter. DMQS information offset is assumed to start at zero.
16	APMInfoAPMStruc. Advanced Power Management BIOS Information
17	APMInfo APMStruc version 1.1. Advanced Power Management BIOS Information
18	SMP_Active DWORD Information on the operating system (OS) support for more than 1 CPU. Returns 1 if the OS has SMP support and 0 if the OS has uniprocessor

support.

19 PSDInfo.psd_flags DWORD PSD status area where several pieces of useful information about the PSD can be obtained. After obtaining the variable address, the caller must test the bit for the desired aspect of the PSD. The PSD flags definition is as follows PSD_INITIALIZED (0x80000000) PSD has been initialized PSD_INSTALLED (0x40000000) PSD has been installed PSD_ADV_INT_MODE (0x20000000) PSD is in advaanced interrupt mode PSD_KERNEL_PIC (0x10000000) Let the kernel interrupt manager EOI

20 cProcessors DWORD Information for the Multi-Processor environment. Indicates the number of processors currently running in the MP environment. A value of 1 is returned in Uni-Processor environment.

GlobalInfoSeg (PSEL)	Address of the global information segment structure, as defined below		
	Time (ULONG)	Time in seconds since 1/1/1970.	
	Millisecs (ULONG)	Time in milliseconds.	
	Hours (UCHAR)	Current hour.	
	Minute (UCHAR)	Current minute.	
	Seconds (UCHAR)	Current second.	
	HundredSec (UCHAR)	Current hundreth of a second.	
	TimeZone (USHORT)	Minutes from UTC. If FFFFH, TimeZone is undefined.	
	Interval (USHORT)	Timer interval in tenths of milliseconds.	
	Day (UCHAR)	Day.	
	Month (UCHAR)	Month.	
	Year (USHORT)	Year.	
	Weekday (UCHAR)	Day of the week	
		0	Sunday
		1	Monday
		2	Tuesday
		3	Wedn

		esday
	4	Thursday
	5	Friday
	6	Saturday
MajorVersion (UCHAR)	Major version number.	
MinorVersion (UCHAR)	Minor version number.	
Revision (UCHAR)	Revision letter.	
CurrentSession (UCHAR)	Current foreground full-screen session.	
MaxNumSessions (UCHAR)	Maximum number of full-screen sessions.	
HugeShift (UCHAR)	Shift count for huge segments.	
ProtModeInd (UCHAR)	Protect-mode-only indicator	
	0	DOS and OS/2 sessions.
	1	OS/2 s

		ession only.
LastProcess (USHORT)	Process ID of the current foreground process.	
DynVarFlag (UCHAR)	Dynamic variation flag	
	0	Absolute.
	1	Enabled.
MaxWait (UCHAR)	Maximum wait in seconds.	
MinTimeSlice (USHORT)	Minimum time slice in milliseconds.	
MaxTimeSlice (USHORT)	Maximum time slice in milliseconds.	
BootDrive (USHORT)	Drive from which the system startup occurred	
	1	Drive A
	2	Drive B
	n	Drive *n*.
TraceFlags (UCHAR)	Thirty-two system trace major code flags. Each bit corresponds to a	

		trace major code 00H-FFH. The most significant bit (left-most) of the first byte corresponds to major code 00H. Values are	
		0	Trace disabled.
		1	Trace enabled.
	MaxTextSessions (UCHAR)	Maximum number of VIO windowable sessions.	
	MaxPMSessions (UCHAR)	Maximum number of Presentation Manager sessions.	
LocalInfoSeg (PSEL)	Address of the selector for the local information segment structure, as defined below		
	ProcessID (PID)	Current Process ID.	
	ParentProcessID (PID)	Parent Process ID.	
	ThreadPrty (USHORT)	Priority of current thread.	
	ThreadID (TID)	Current Thread ID.	
	SessionID (USHORT)	Current Session ID.	
	ProcStatus (UCHAR)	Process status.	
	Unused (UCHAR)	Unused.	
	ForegroundProcess (BOOL)	Current process is in foreground (has keyboard focus)	
		1	Imp

		lies YES
	0	Implies NO.
TypeProcess (UCHAR)	Type of process	
	0	Full screen protect-mode session.
	1	Requires real mode.
	2	

3	VIO windowable protect-mode session.
4	Presentation Manager protect-mode session.

		Detached protect-mode process.
	Unused (UCHAR)	Unused.
	EnvironmentSel (SEL)	Environment selector.
	CmdLineOff (USHORT)	Command line offset in the segment addressed by EnvironmentSel.
	DataSegLen (USHORT)	Length of the data segment in bytes.
	StackSize (USHORT)	Stack size in bytes.
	HeapSize (USHORT)	Heap size in bytes.
	HModule (UHMODULE)	Module handle.
	DSSel (SEL)	Data segment selector.
APMInfo	Advanced Power Management BIOS Information, as defined below	
	APM CodeSeg (WORD)	APM 16-bit code segment (real-mode segment base address). From APM BIOS, INT 15h AX
	APM DataSeg (WORD)	APM 16-bit data segment (real-mode segment base address). From APM BIOS, INT 15h AX
	APM Offset (WORD)	Offset to entry point. From APM BIOS, INT 15h AX
	APM Flags (WORD)	APM capability

	flags. From APM BIOS, INT 15h AX
APM Level (WORD)	APM revision level. From APM BIOS, INT 15h AX
APM CPUIdle (6 bytes (DF))	APM Services Entry Point for CPU Idle and Busy Functions.

Note: APM CodeSeg and APM DataSeg are segment addresses, not selectors. It is the responsibility of the device driver to convert the segment address to a valid protect-mode selector.

The first time GetDOSVar is called at device-driver initialization with index (AL) = 10H and CX = 1, the system sets the values for APM CodeSeg, APM DataSeg, APM Offset, APM Flags, and APM Level. On return, AX BX points to the APMInfo structure.

If GetDOSVar is called at device-driver initialization with index (AL) = 10h and CX = 0, the system sets the values for APM Flags and APM Level. On return, AX BX points to the APMInfo structure. Other fields in the APMInfo structure might have been set by a previous call to GetDOSVar with index = 10h and CX = 1.

If GetDOSVar is called after device-driver initialization with index (AL) = 10H, no information in the APMInfo structure is modified. On return, AX BX points to the APMInfo structure.

APM CPUIdle contains the address of the CPU Idle and Busy processing routines from the Power Management Services device driver. This variable is initially empty (NULL) until Power Management Services loads and places the addresses for the CPU Idle and Busy routines into the variable area. The variable address must be the 16 16 Selector Offset. The Offset is 0-extended to 32 bits, and the value must be represented in 16 32 format. The APM CPUIdle function utilizes the AX register as the *control selection* flag for BUSY (AX=00001H) and IDLE (AX=0000H) requests.

These variables are maintained by the kernel for the benefit of physical device drivers. Notice that the address returned is the address of the indicated variable; the variable can contain a vector to some facility, or it can contain a structure.

DevHlp_KillProc

DevHlp_KillProc kills a process unconditionally.

Calling Sequence in Assembler

```
MOV   BX, pid
MOV   DL, DevHlp_KillProc

CALL  [Device_Help]
```

Results in Assembler

C Clear if the process is killed. AX = zero.

C Set if error. AX = Error code.

Possible errors

217 **ERROR_ZOMBIE_PROCESS** Process is already dead pending collection of result codes by parent.

303 **ERROR_INVALID_PROCID** PID is invalid.

Calling Sequence in C

There is no direct C calling Sequence.

Remarks

The process is killed unconditionally. Signal and exception handlers are not run. This Device Help may be called at Task Time only.

DevHlp_OpenFile

DevHlp_OpenFile is used by base device drivers to open a file for read access during initalization.

Calling Sequence in Assembler

```
LES   DI, FileOpen            ; Point to FILEIOINFO structure
MOV   DL, DevHlp_OpenFile

CALL   [Device_Help]
```

ES DI points to a FILEIOINFO structure defined as follows

```
FILEIOINFO struc
length        dw     8    ; length of imbedded fle system operation structure
;
FOPEN struc
name          dd     ?    ; 1616 pointer to ASCIZ pathname
fsize         dd     ?    ; returned size of file
FOPEN         ends
;
FILEIOINFO    ends
```

Results in Assembler

C Clear if file is opened. AX = zero.

C Set if error. AX = Error code. Possible errors

24 **ERROR_BAD_LENGTH**	The length in the FILEIOINFO structure is invalid.

Calling Sequence in C

```
#include  "dhcalls.h"

USHORT APIENTRY DevHelp_OpenFile (PFILEIOINFO pFileOpen);
```

USHORT APIENTRY DevHlp_OpenFile (PFILEIOINFO pFileOpen)

pFILEIOINFO   input   Pointer to the FILEIOINFO structure defined as follows

```
typedef struct FOPEN {
        PSZ    FileName;   /* (input) pointer to file name */
        ULONG  FileSize;   /* (output) size of file returned by FileOp
} FILEOPEN;

typedef struct FCLOSE {
        USHORT  reserved   /* reserved */
} FILECLOSE;

typedef struct FREAD {
        PBYTE Buffer;     /* (input) pointer to input buffer */
        ULONG ReadSize;   /* (input) number of bytes to read fromfile
} FILEREAD;

typedef struct FREADAT {
        PBYTE Buffer;        /* (input) pointer to input buffer */
        ULONG ReadSize;      /* (input) number of bytes to read from f
        ULONG StartPosition  /* (input) starting file position relativ
                                  the beginning of the file */
} FILEREADAT;

typedef union FILEIOOP {
        struct FOPEN FileOpen;
        struct FCLOSE FileClose;
        struct FREAD FileRead;
        struct FREADAT FileReadAt;
} FILEIOOP;

typedef struc _DDFileIo {
        USHORT Length; /* (input) length of imbedded structure */
        FILEIOOP Data; /* (input) imbedded file system operation struc
} FILEIOINFO, FAR * PFILEIOINFO
```

Results in C

Success Indicator 0 if file was opened..

24 **ERROR_BAD_LENGTH**	Length in the FILEIOINFO structure is invalid.

Remarks

DevHlp_OpenFile may be called at initialization time only. It provides a primitive interface to the mini-IFS or micro_IFS at initialization time.

Drive and path information is ignored. The system searches for the file in the root, \OS2 and \OS2\BOOT directories of the boot drive/device.

Using this interface, one file only may be opened at a time. No handle is assigned. Subsequent read and close requests assume the file is the most recent opened using DevHlp_OpenFile.

DevHlp_PerfSysTrace

DevHlp_PerfSysTrace writes software trace information to the STRACE buffer.

Calling Sequence in Assembler

```
MOV   AX, MajorCode
MOV   BX, TraceSize
MOV   CX, MinorCode
LDS   SI, TraceData
MOV   DL, DevHlp_PerfSysTrace
CALL  DevHlp
JC    Error
```

Results in Assembler

AX = return code.

Possible values

0 **NO_ERROR**	Data written to trace buffer.
32902 **ERROR_NOMEMORY**	Trace buffer has not been allocated.

Calling Sequence in C

```
#include "dhcalls.h"
```

USHORT APIENTRY DevHelp_PerfSysTrace (USHORT Major, USHORT Minor,USHORT TraceSize, PBYTE TraceData)

Remarks

A trace buffer must be allocated, wia the STRACE INIT command, before attempting to write trace data.

Tracing stops once the trace buffer fills up. No error indication is returned. Subsequent calls to DevHelp_PerfSysTrace return immediately without writing any data.

DevHlp_QSysState

DevHlp_QSysState is used by physical device drivers to obtain system status information..

CallingSequence in Assembler

```
MOV   EAX, EntityList
MOV   EBX, EntityLevel
MOV   EDI, pidtid
MOV   ESI, pDataBuf
MOV   ECX, cbDataBuf
MOV   DL,  DevHlp_QSysState
CALL  [Device_Help]
```

Results in Assembler

'C' Clear if process killed. AX = zero.

'C" Set if error.

Possible errors

87 **ERROR_INVALID_PARAMETER** Invalid parameter specified.

111 **ERROR_BUFFER_OVERFLOW** Data buffer is too small to hold all returned information.

115 **ERROR_PROTECTION_VIOLATION** Unable to store in to data buffer.

124 **ERROR_INVALID_LEVEL** Data buffer is too small to hold all returned information.

Calling Sequence in C

There is no direct C calling Sequence.

Remarks

DevHlp_QSysState is functionally equivalent to DosQuerySysState. See DosQuerySysState for information on the entities that may be requested and the format of the entities returned. This Device Help may be call at Task Time only.

DevHlp_ReadFile

DevHlp_ReadFile is used by base device drivers to read a file previously opened using DevHlp_OpenFile.

Calling Sequence in Assembler

```
LES   DI, ReadFile
MOV   DL, DevHlp_ReadFile

CALL  [Device_Help]
```

ES DI points to a FILEIOINFO structure defined as follows

```
FILEIOINFO struc
length        dw     8    ; length of imbedded fle system operation structure
;
FREAD struc
Buffer        dd     ?    ; 1616 pointer to the input buffer
ReadSize      dd     ?    ; length of data to read
FREAD         ends
;
FILEIOINFO    ends
```

Results in Assembler

C Clear if the file is closed. AX = zero.

C Set if error. Possible errors

24 **ERROR_BAD_LENGTH**	The length in the FILEIOINFO structure is invalid.

Calling Sequence in C

```
#include  "dhcalls.h"
```

USHORT APIENTRY DevHelp_ReadFile (PFILEIOINFO pfileread)

Pointer to the FILEIOINFO structure defined as follows

```
typedef struct FOPEN {
        PSZ    FileName;   /* (input) pointer to file name */
        ULONG  FileSize;   /* (output) size of file returned by FileOpen */
} FILEOPEN;

typedef struct FCLOSE {
        USHORT  reserved   /* reserved */
} FILECLOSE;

typedef struct FREAD {
        PBYTE Buffer;      /* (input) pointer to input buffer */
        ULONG ReadSize;    /* (input) number of bytes to read fromfile */
} FILEREAD;

typedef struct FREADAT {
        PBYTE Buffer;        /* (input) pointer to input buffer */
        ULONG ReadSize;      /* (input) number of bytes to read from file */
        ULONG StartPosition  /* (input) starting file position relative to
                                 the beginning of the file */
} FILEREADAT;

typedef union FILEIOOP {
        struct FOPEN FileOpen;
        struct FCLOSE FileClose;
        struct FREAD FileRead;
        struct FREADAT FileReadAt;
} FILEIOOP;

typedef struc _DDFileIo {
        USHORT Length; /* (input) length of imbedded structure */
        FILEIOOP Data; /* (input) imbedded file system operation structure */
} FILEIOINFO, FAR * PFILEIOINFO
```

Results in C

Success Indicator 0 if file was read.

24 **ERROR_BAD_LENGTH**	Length in the FILEIOINFO structure is invalid.

Remarks

DevHlp_FileRead may be called at initialization time only. It provides a primitive interface to the mini-IFS or micro_IFS at initialization time.

The file is read from the current file position. After a successful read, the current file position is updated.

Using this interface only one file may be opened at a time. No handle is assigned by open. The file read is assumed to be the most recent opened using DevHlp_OpenFile.

DevHlp_ReadFileAt

DevHlp_ReadFileAt is used by base device drivers to read a file previously opened using DevHlp_OpenFile from a specified file location.

Calling Sequence in Assembler

```
LES   DI, ReadFileAt
MOV   DL, DevHlp_ReadFileAt

CALL  [Device_Help]
```

ES DI points to a FILEIOINFO structure defined as follows

```
FILEIOINFO struc
length        dw   12    ; length of imbedded fle system operation structure
;
FREADAT struc
Buffer        dd   ?    ; 1616 pointer to input buffer
Readsize      dd   ?    ; length of data to read
StartPosition dd   ?    ; starting position relative to the beginning of the file
FREADAT       ends
;
FILEIOINFO    ends
```

Results in Assembler

C Clear if file is closed. AX = zero.

C Set if error. AX = Error code. Possible errors

24 **ERROR_BAD_LENGTH** The length in the FILEIOINFO structure is invalid.

Calling Sequence in C

```
#include  "dhcalls.h"
```

USHORT APIENTRY DevHelp_FileReadAt (PFILEIOINFO pFileReadAt)

Pointer to the FILEIOINFO structure defined as follows

```
typedef struct FOPEN {
        PSZ    FileName;   /* (input) pointer to file name */
        ULONG  FileSize;   /* (output) size of file returned by FileOpen */
} FILEOPEN;

typedef struct FCLOSE {
        USHORT  reserved   /* reserved */
} FILECLOSE;

typedef struct FREAD {
        PBYTE Buffer;     /* (input) pointer to input buffer */
        ULONG ReadSize;   /* (input) number of bytes to read from file */
} FILEREAD;

typedef struct FREADAT {
        PBYTE Buffer;        /* (input) pointer to input buffer */
        ULONG ReadSize;      /* (input) number of bytes to read from file */
        ULONG StartPosition  /* (input) starting file position relative to
                                    the beginning of the file */
} FILEREADAT;

typedef union FILEIOOP {
        struct FOPEN FileOpen;
        struct FCLOSE FileClose;
        struct FREAD FileRead;
        struct FREADAT FileReadAt;
} FILEIOOP;

typedef struc _DDFileIo {
        USHORT Length; /* (input) length of imbedded structure */
        FILEIOOP Data; /* (input) imbedded file system operation structure */
} FILEIOINFO, FAR * PFILEIOINFO
```

Results in C

Success Indicator 0 if file was read.

24 **ERROR_BAD_LENGTH**	Length in the FILEIOINFO structure is invalid.

Remarks

DevHlp_ReadFileAt may be called at initialization time only. It provides a primitive interface to the mini-IFS or micro_IFS at initialization time.

The current file position is set according to the StartPosition. The file is read from that file position. After a successful read, the current file position is updated.

Using this interface, one file only may be opened at a time. No handle is assigend by open. The file read is assumed to be the most recent opened using DevHlp_OpenFile.

DevHlp_SysTrace

DevHlp_SysTrace function provides a service for device drivers to add information to the System Trace buffer.

Calling Sequence in Assembler

```
MOV    AX, Major Code
MOV    BX, Length
MOV    CX, Minor Code

LDS    SI, Data

MOV    DL, 28H

CALL   [Device_Help]
```

Results in Assembler

If CF = 0, trace record placed in trace buffer

Else data not traced.

Possible errors

- Tracing suspended
- Minor code not being traced
- PID not being traced
- Trace overrun

Calling Sequence in C

```
#include  "dhcalls.h"
```

USHORT APIENTRY DevHlp_SysTrace (USHORT Major, USHORT Minor, USHORT Size, PBYTE Datar)

Major (USHORT)	Major trace event code (240 255).
Minor (USHORT)	Minor trace event code (0 255).
Size(USHORT)	Length of the variable length area to be recorded (0 512).
Data (PBYTE)	Pointer to the area to be traced.

Results in C

Success indicator 0.

Possible errors

- Data not traced, e.g., major event code is not currently selected for tracing.

Remarks

The trace facility maintains an array of 32 bytes (256 bits), in which each bit represents a major event code. This array is updated each time the user enables or disables tracing of a major event. The device driver must check this array before calling DevHlp_SysTrace to ensure that the major event specified is currently enabled for tracing. This array is located in the Global InfoSegAll registers are preserved.

Interrupts are disabled while the trace data is saved and then re-enabled if they were initially enabled.

DevHlp_SysTrace is synonymous with DevHlp_RAS.

Control Program Functions

This chapter contains an alphabetic list of the following Control Program functions.

- DosAliasMem
- DosCancelLockRequestL
- DosClose
- DosCreateEventSem
- DosCreateThread2
- DosDumpProcess
- DosFindFirst
- DosFindNext
- DosForceSystemDump
- DosGetProcessorStatus
- DosListIO
- DosListIOL
- DosOpen
- DosOpenL
- DosPerfSystemCall
- DosProtectOpenL
- DosProtectQueryFileInfo
- DosProtectSetFileInfo
- DosProtectSetFileLocksL
- DosProtectSetFilePrtL
- DosProtectSetFileSizeL
- DosQueryABIOSSuport
- DosQueryFileInfo
- DosQueryMemState
- Dos16QueryModFromCS
- DosQueryModFromEIP
- DosQueryPathInfo

- DosQuerySysInfo
- DosQuerySysState
- DosQueryThreadAffinity
- DosRead
- DosReplaceModule
- DosSetFileInfo
- DosSetFileLocksL
- DosSetFilePtr
- DosSetFilePtrL
- DosSetFileSizeL
- DosSetPathInfo
- DosSetProcessorStatus
- DosSetThreadAffinity
- Dos16SysTrace
- DosTmrQueryFreq
- DosTmrQueryTime
- DosVerifyPidTid
- DosWrite

The following APIs, from the list above, are Raw File System APIs.

- DosClose
- DosListIO
- DosOpen
- DosRead
- DosSetFilePtr
- DosWrite

The OS/2 raw file system provides an interface for applications to manage data efficiently on the logical partitions or physical hard drives installed in a system. Some of the raw file system function is available by using a combination of the DosPhysicalDisk and DosDevIOCtl application programming interfaces.

The OS/2 raw file system provides a programming abstraction that treats each logical partition or physical disk as one large file that can be opened, locked, seeked, read from, written to, and closed. Logical partitions are identified using the Universal Naming Convention (UNC) in the form of '\\.\X ', where 'X' can be substituted with the letter corresponding the logical partition desired on any hard drive, floppy disk or CD-ROM drive. Physical disks are identified using UNC naming in the form of '\\.\Physical_Disk#', where '#' is replaced with the physical disk number corresponding to the number found in the LVM command. The combination of the naming convention and use of the common file system application programming interfaces (APIs) provides a greatly simplified migration path for applications.

Traditionally, raw file systems have been utilized by applications that manage large amounts of data under heavy workloads. Typically, this has been commercial database servers performing on-line transaction processing. Disk I/O can become a bottleneck under these conditions and the use of an efficient raw file system can be very useful in improving system performance, through reduced path length and serialization.

DosAliasMem

Purpose

DosAliasMem creates a private Read/Write alias or an LDT code segment alias to part of an existing memory object. The alias object is accessible only to the process that created it. The original object must be accessible to the caller of DosAliasMem.

Syntax

```
#define INCL_DOSMEMMGR
#include os2.h>
```

APIRET APIENTRY DosAliasMem **(PVOID pMem, ULONG cbSize, PPVOID ppAlias, ULONG flags)**

Parameters

pMem (PMEM)   input

Contains the address of the memory to be aliased. It must be on a page boundary (that is, 4K aligned), but may specify an address within a memory object.

cbSize (CBSIZE)   input

Specifies the size in bytes for the memory to alias. The entire range must lie within a single memory object and must be committed if OBJ_SELMAPALL is specified.

ppAlias (PPALIAS)   output

Address of a location in which the address of the aliased memory is returned. The corresponding LDT selector is not explicitly returned but may be calculated by using the Compatibility Mapping Algorithm

```
sel = (SEL) ((ULONG) (*ppAlias) >> 13 | 7)
```

flags (FLAGS)   input

Flags are defined as follows

OBJ_SELMAPALL (0x00000800)	OBJ_SELMAPALL creates a Read/Write 32 bit alias to the address specified. The entire range must be committed, start on page boundary and be within the extent of a single memory object. An LDT selector is created to map the entire range specified. If OBJ_SELMAPALL is not specified, then size is rounded up to a 4K multiple and the alias created inherits the permissions from the pages of the original object. OBJ_TILE may be specified, but currently this is enforced whether or not specified. This forces LDT selectors to be based on 64K boundaries.
SEL_CODE (0x00000001)	Marks the LDT alias selector(s) Read-Executable code selectors.
SEL_USE32 (0x00000002)	Used with OBJ_SELMAPALL, otherwise ignored. Marks the first alias LDT selector as a 32 bit selector by setting the BIG/C32 bit.

Returns

ulrc (APIRET)   returns

Return Code.

DosAliasMem returns one of the following values

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
32798	ERROR_CROSSES_OBJECT_BOUNDARY

Remarks

An export for DosAliasMem does not appear in versions of OS2386.LIB distributed prior to Warp Server for e-business. When using older versions, the following statements should be added to the link edit .DEF file

```
imports
DosAliasMem = DOSCALLS.298
```

An alias is removed by calling DosFreeMem with the alias address.

Though it is possible to create a Read/Write alias to a code segment to allow code modification this is not recommended. On Pentium(R) processors, and later, pipe-lining techniques used by the processor might allow the processor not to be aware of the modified code, if appropriate pipe-line serialization is not performed by the programmer. For further information see the processor documentation.

Related Functions

- DosAllocMem
- DosAllocSharedMem
- DosFreeMem

Example Code

```
#define INCL_DOSMEMMGR
#include int main(int argc, char *argv[], char *envp[])
{
   PVOID pAlias;
   PVOID pMem;
   APIRET rc;

   /* alias a read-only shared memory object as a private read/write */
   /* object. This will allow clients of this object to read only    */
   /* while allowing the owner to update it.                         */

   rc=DosAllocSharedMem(pMem,NULL,128*1024,
      PAG_READ+PAG_COMMIT+OBJ_GIVEABLE);

   rc = DosAliasMem(pMem, 128*1024, pAlias, OBJ_TILE);

   .
   .
   .
   .

   return 0;
}
```

DosCancelLockRequestL

Purpose

DosCancelLockRequestL cancels an outstanding DosSetFileLocksL request.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosCancelLockRequestL **(HFILE hFile, PFILELOCKL pflLock)**

Parameters

hFile HFILE)   input
File handle used in the DosSetFileLocksL function that is to be cancelled.

pflLockL PFILELOCKL)   input
Address of the structure describing the lock request to cancel.

Returns

ulrc APIRET)   returns
Return Code.

DosCancelLockRequestL returns one of the following values

0	NO_ERROR
6	ERROR_INVALID_HANDLE

87	ERROR_INVALID_PARAMETER
173	ERROR_CANCEL_VIOLATION

Remarks

DosCancelLockRequestL allows a process to cancel the lock range request of an outstanding DosSetFileLocksL function.

If two threads in a process are waiting on a lock file range, and another thread issues DosCancelLockRequestL for that lock file range, then both waiting threads are released.

Not all file-system drivers (FSDs) can cancel an outstanding lock request.

Local Area Network (LAN) servers cannot cancel an outstanding lock request if they use a version of the operating system prior to OS/2 Version 2.00.

Related Functions

- DosSetFileLocksL

Example Code

This example opens a file named CANLOCK.DAT , locks a block of the data, writes some data to it, and then cancels the lock request.

```
#define INCL_DOSFILEMGR       /* File Manager values */
#define INCL_DOSERRORS        /* DOS Error values    */
#include os2.h
#include stdio.h
#include string.h

int main(VOID)

HFILE     FileHandle   = NULLHANDLE;  /* File handle */
ULONG     Action       = 0,           /* Action taken by DosOpenL */
Wrote        = 0;           /* Number of bytes written by DosWrite */
CHAR      FileData40 = "Forty bytes of demonstration text data\r\n";
APIRET    rc           = NO_ERROR;    /* Return code */

FILELOCKL  LockArea     = 0,         /* Area of file to lock */
UnlockArea   = 0;         /* Area of file to unlock */

rc = DosOpenL("canlock.dat",                 /* File to open */
FileHandle,                   /* File handle */
Action,                       /* Action taken */
256,                           /* File primary allocation */
FILE_ARCHIVED,                 /* File attributes */
FILE_OPEN | FILE_CREATE,       /* Open function type */
OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
0L);                           /* No extended attributes */
if (rc != NO_ERROR)                        /* If open failed */
printf("DosOpenL error return code = %u\n", rc);
return 1;

LockArea.lOffset = 0;               /* Start locking at beginning of file */
LockArea.lRange =  40;              /* Use a lock range of 40 bytes       */
UnLockArea.lOffset = 0;             /* Start unlocking at beginning of file */
UnLockArea.lRange =  0;             /* Use a unlock range of 0 bytes       */

rc = DosSetFileLocksL(FileHandle,        /* File handle   */
UnlockArea,       /* No unlock area */
LockArea,         /* Lock current record */
2000L,             /* Lock time-out value of 2 seconds */
0L);               /* Exclusive lock, not atomic */
if (rc != NO_ERROR)
printf("DosSetFileLocks error return code = %u\n", rc);
return 1;


rc = DosWrite(FileHandle, FileData, sizeof(FileData), Wrote);
if (rc != NO_ERROR)
printf("DosWrite error return code = %u\n", rc);
return 1;

/* Should check if (rc != NO_ERROR) here... */

LockArea.lOffset = 0;               /* Start locking at beginning of file */
LockArea.lRange =  0;               /* Use a lock range of 40 bytes       */
UnLockArea.lOffset = 0;             /* Start locking at beginning of file */
UnLockArea.lRange = 40;             /* Use a lock range of 40 bytes       */

rc = DosSetFileLocksL(FileHandle,        /* File handle   */
UnlockArea,       /* Unlock area */
LockArea,         /* No Lock */
2000L,             /* Lock time-out value of 2 seconds */
0L);               /* Exclusive lock, not atomic */
if (rc != NO_ERROR)
printf("DosSetFileLocksL error return code = %u\n", rc);
return 1;
rc = DosClose(FileHandle);
/* Should check if (rc != NO_ERROR) here... */

return NO_ERROR;
rc = DosClose(FileHandle);
/* Should check if (rc != NO_ERROR) here... */

return NO_ERROR;
```

DosClose

Purpose

DosClose closes a handle to a disk.

Syntax

```
#define INCL_DOSFILEMGR
#include os2.h>
```

APIRET DosClose **(HFILE hFile)**

Parameters

hFile (HFILE) input
The handle returned by DosOpen.

Returns

ulrc (APIRET) returns
Return Code.

DosClose returns one of the following values

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE

Remarks

The disk can no longer be accessed using this handle. If opened with the OPEN_SHARE_DENYREADWRITE flag, the disk is unlocked.

Related Functions

- DosOpen

Example Code

The following is NOT a complete usable program. It is simply intended to provide an idea of how to use Raw I/O File System APIs (e.g. DosOpen, DosRead, DosWrite, DosSetFilePtr, and DosClose).

This example opens physical disk #1 for reading and physical disk #2 for writing. DosSetFilePtr is used to set the pointer to the beginning of the disks. Using DosRead and DosWrite, 10 megabytes of data is transferred from disk #1 to disk #2. Finally, DosClosed is issued to close the disk handles.

It is assumed that the size of each of the two disks is at least 10 megabytes.

```
#define INCL_DOSFILEMGR          /* Include File Manager APIs */
#define INCL_DOSMEMMGR           /* Includes Memory Management APIs */
#define INCL_DOSERRORS           /* DOS Error values */
#include os2.h>
#include stdio.h>
#include string.h>

#define SIXTY_FOUR_K 0x10000
#define ONE_MEG     0x100000
#define TEN_MEG     10*ONE_MEG

#define UNC_DISK1  "\\\\.\\Physical_Disk1"
#define UNC_DISK2  "\\\\.\\Physical_Disk2"

int main(void) {
   HFILE  hfDisk1        = 0;      /* Handle for disk #1 */
   HFILE  hfDisk2        = 0;      /* Handle for disk #2 */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   ULONG  cbRead         = 0;      /* Bytes to read */
   ULONG  cbActualRead   = 0;      /* Bytes read by DosRead */
   ULONG  cbWrite        = 0;      /* Bytes to write */
   ULONG  ulLocation     = 0;
   ULONG  cbActualWrote  = 0;      /* Bytes written by DosWrite */
   UCHAR  uchFileName1[20]  = UNC_DISK1, /* UNC Name of disk 1 */
          uchFileName2[20]  = UNC_DISK2; /* UNC Name of disk 2 */
   PBYTE  pBuffer        = 0;
   ULONG  cbTotal        = 0;

   APIRET rc             = NO_ERROR;            /* Return code */

   /* Open a raw file system disk #1 for reading */
   rc = DosOpen(uchFileName1,               /* File name */
                hfDisk1,                   /* File handle */
                ulAction,                  /* Action taken by DosOpen */
                0L,                         /* no file size */
                FILE_NORMAL,                /* File attribute */
                OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
                OPEN_SHARE_DENYNONE |       /* Access mode */
                OPEN_ACCESS_READONLY,
                0L);                        /* No extented attributes */
   if (rc != NO_ERROR) {
      printf("DosOpen error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Set the pointer to the begining of the disk */
   rc = DosSetFilePtr(hfDisk1,      /* Handle for disk 1 */
                      0L,           /* Offset must be multiple of 512 */
                      FILE_BEGIN,   /* Begin of the disk */
                      ulLocation); /* New pointer location */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Open a raw file system disk #2 for writing */
   rc = DosOpen(uchFileName2,               /* File name */
                hfDisk2,                   /* File handle */
                ulAction,                  /* Action taken by DosOpen */
                0L,                         /* no file size */
                FILE_NORMAL,                /* File attribute */
                OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
                OPEN_SHARE_DENYNONE |       /* Access mode */
                OPEN_ACCESS_READWRITE,
                0L);                        /* No extented attributes */
   if (rc != NO_ERROR) {
      printf("DosOpen error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Set the pointer to the begining of the disk */
   rc = DosSetFilePtr(hfDisk2,      /* Handle for disk 1 */
                      0L,           /* Offset must be multiple of 512 */
                      FILE_BEGIN,   /* Begin of the disk */
                      ulLocation); /* New pointer location */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error rc = %u\n", rc);
      return(1);
   } /* endif */
   /* Allocate 64K of memory for transfer operations */
   rc = DosAllocMem((PPVOID)pBuffer, /* Pointer to buffer */
                     SIXTY_FOUR_K,      /* Buffer size */
                     PAG_COMMIT |     /* Allocation flags */
                     PAG_READ |
                     PAG_WRITE);
   if (rc != NO_ERROR) {
      printf("DosAllocMem error rc = %u\n", rc);
      return(1);
   } /* endif */

   cbRead = SIXTY_FOUR_K;
   while (rc == NO_ERROR  cbTotal  TEN_MEG) {

      /* Read from #1 */
      rc = DosRead(hfDisk1,         /* Handle for disk 1 */
                   pBuffer,         /* Pointer to buffer */
                   cbRead,          /* Size must be multiple of 512 */
                   cbActualRead);  /* Actual read by DosOpen */
      if (rc) {
         printf("DosRead error return code = %u\n", rc);
         return 1;
      }

      /* Write to disk #2 */
      cbWrite = cbActualRead;
      rc = DosWrite(hfDisk2,         /* Handle for disk 2 */
                    pBuffer,         /* Pointer to buffer */
                    cbWrite,         /* Size must be multiple of 512 */
                    cbActualWrote); /* Actual written by DosOpen */
      if (rc) {
         printf("DosWrite error return code = %u\n", rc);
         return 1;
      }
      if (cbActualRead != cbActualWrote) {
         printf("Bytes read (%u) does not equal bytes written (%u)\n",
                cbActualRead, cbActualWrote);
         return 1;
      }
      cbTotal += cbActualRead; /* Update total transferred */
   }

   printf("Transfer successfully %d bytes from disk #1 to disk #2.\n",
          cbTotal);

   /* Free allocated memmory */
   rc = DosFreeMem(pBuffer);
   if (rc != NO_ERROR) {
      printf("DosFreeMem error return code = %u\n", rc);
      return 1;
   }

   rc = DosClose(hfDisk1);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
   }

   rc = DosClose(hfDisk2);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
   }
return NO_ERROR;
}
```

DosCreateEventSem

Purpose

DosCreateEventSem creates an event semaphore.

Syntax

```
#define INCLDOSSEMAPHORES
#include os2.h
```

APIRET DosCreateEventSem **(PSZ pszName, PHEV phev, ULONG flAttr, BOOL32 fState)**

Parameters

pszName PSZ)   input

A pointer to the ASCIIZ name of the semaphore.

Semaphore names are validated by the file system, and must include the prefix \SEM32\. A maximum of 255 characters is allowed. If these requirements are not met, ERROR_INVALID_NAME is returned. If the semaphore already exists, ERROR_DUPLICATE_NAME is returned.

If this field is null, the semaphore is unnamed. Unnamed event semaphores can be either private or shared, depending on *flAttr* . They are identified by the semaphore handle that *phev* points to.

By default, all named semaphores are shared.

phev PHEV)   output

A pointer to the handle of the event semaphore.

flAttr ULONG)   input

A set of flags that specify the attributes of the event semaphore.

DC_SEM_SHARED	If the DC_SEM_SHARED bit is set, the semaphore is shared. Otherwise, this flag should be set to 0L. This bit is checked only if the semaphore is unnamed (that is, if *pszName* is null), because all named semaphores are shared.
DCE_AUTORESET (0x1000)	Causes the semaphore to be reset automatically at the time it is posted. 0x1000
DCE_POSTONE (0x0800)	Causes one thread only to be posted where multiple threads are waiting on an event semaphore created with this attribut. DCE_POSTONE also causes the semaphore to be reset automatically when it is posted.

fState BOOL32)   input

Initial state of the semaphore.

Possible values are defined in the list below

0	FALSE The initial state of the semaphore is reset.
1	TRUE The initial state of the semaphore is posted.

Returns

ulrc APIRET)   returns

Return Code.

DosCreateEventSem returns one of the following values

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
123	ERROR_INVALID_NAME
285	ERROR_DUPLICATE_NAME
290	ERROR_TOO_MANY_HANDLES

- DosCloseEventSem
- DosOpenEventSem
- DosPostEventSem

- DosQueryEventSem
- DosResetEventSem
- DosWaitEventSem

Example Code

This example creates an event semaphore, and the asynchronous timer posts to it when its time interval expires. Finally, the event semaphore is closed.

```
#define INCL_DOSSEMAPHORES   /* Semaphore values */
#define INCL_DOSDATETIME     /* Timer support    */
#define INCL_DOSERRORS       /* DOS error values */
#include os2.h
#include stdio.h

int main(VOID)

PSZ     szSemName  = "\\SEM32\\TIMER\\THREAD1\\EVENT1"; /* Semaphore name */
HEV     hevEvent1     = 0;                   /* Event semaphore handle    */
HTIMER  htimerEvent1  = 0;                   /* Timer handle              */
APIRET  rc            = NO_ERROR;            /* Return code               */

rc = DosCreateEventSem(szSemName,      /* Name of semaphore to create  */
hevEvent1,     /* Handle of semaphore returned */
DC_SEM_SHARED,  /* Shared semaphore             */
FALSE);         /* Semaphore is in RESET state  */
if (rc != NO_ERROR)
printf("DosCreateEventSem error return code = %u\n", rc);
return 1;

rc = DosAsyncTimer(7000L,    /* 7 second interval            */
(HSEM) hevEvent1,            /* Semaphore to post            */
htimerEvent1);              /* Timer handler (returned)     */
if (rc != NO_ERROR)
printf("DosAsyncTimer error return code = %u\n", rc);
return 1;
 else
printf("Timer will expire in about 7 seconds...\n");


/* ... add your other processing here... */

rc = DosWaitEventSem(hevEvent1,            /* Wait for AsyncTimer event */
(ULONG) SEM_INDEFINITE_WAIT);              /* As long as it takes       */
if (rc != NO_ERROR)
printf("DosWaitEventSem error return code = %u\n", rc);
return 1;


rc = DosCloseEventSem(hevEvent1);      /* Get rid of semaphore       */
if (rc != NO_ERROR)
printf("DosCloseEventSem error return code = %u", rc);
return 1;


return NO_ERROR;
```

DosCreateThread2

Purpose

DosCreateThread2 creates an asynchronous thread of execution under the current process using a pre-allocated stack.

Syntax

```
#define INCL_DOSPROCESS
#include os2.h>
```

APIRET DosCreateThread2 **(PTHREADCREATE ptc, ULONG cbSize, PTID pTid, PFNTHREAD pfnStart, ULONG lParam, ULONG lFlag, PBYTE pStack, ULONG cbStack)**

Parameters

ptc(PTHREADCREATE)   input/output

Address of the thread create data structure

```
typedef struct_THREADCREATE{
   ULONG       cbSize;
   PTID        pTid;
   PFNTHREAD   pfnStart;
   ULONG       lParam;
   ULONG       lFlag;
   PBYTE       pStack;
   ULONG       cbStack;
}  THREADCREATE;
typedef THREADCREATE *PTHREADCREATE;
```

cbSize (ULONG)   input

The size, in bytes, of the thread create structure.

pTid (PTID)   output

The thread identifier of the created thread is returned.

pfnStart (PFNTHREAD)   input

Address of the code to be executed when the thread begins execution.

This function is called using a 32 bit near-call, accepts a single parameter, lParam, and returns a doubleword exit status (see DosExit). Returning from the function without executing DosExit causes the thread to end. In this case, the exit status is the value in the EAX register when the thread ends.

lParam (ULONG)   input

An argument that is passed to the target thread routine as a parameter. It is usually a pointer to a parameter block.

lFlag (ULONG)   input

Thread flags.

Possible values are a combination of the following

CREATE_READY (0x00000000)   The new thread starts immediately.

CREATE_SUSPEND (0x00000001) The thread is created in the suspended state, and the creator of the thread must issue DosResumeThread to start the new thread's execution.

STACK_SPARSE (0x00000000)   The system uses the default method for initializing the thread's stack.

STACK_COMMITTED (0x00000002) The system precommits all the pages in the stack. One page is 4KB

pStack (PBYTE)   input

Address of the top of the stack (not the bottom of the stack).

cbStack (ULONG)   input

The size, in bytes, of the new thread's stack.

Returns

ulrc (APIRET)   returns

Return Code.

DosCreateThread2 returns one of the following values

0	NO_ERROR
8	ERROR_NOT_ENOUGH_MEMORY
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
115	ERROR_PROTECTION_VIOLATION
164	ERROR_MAX_THRDS_REACHED

Remarks

The difference between DosCreateThread and DosCreateThread2 is the use and management of the thread's stack.

Using DosCreateThread, the application is not responsible for allocating the stack. The application simply supplies the size of the stack, and the operating system will manage the allocation and location of that storage on behalf of the application. DosCreateThread also employs guard pages and exception handling for stack related situations, such as stack growth. One of the problems with DosCreateThread is that for each thread, a minimum 64K of virtual address space is reserved, but only 8K of physical storage is actually committed. Therefore, 56K of virtual address space is wasted initially.

Using DosCreateThread2, the application is responsible for allocating the stack before calling the API. With DosCreateThread2, the operating system gives the application total control over stack size and location. Therefore, more efficient use of virtual address space within the system can be achieved.

The address of the stack, **pStack**, must be in the compatibility region, that is, the first 512MB (0x20000000). If DosCreateThread2 is called with a stack address higher than 512MB, ERROR_INVALID_PARAMETER will be returned.

Related Functions

- DosCreateThread

Example Code

In this example, the main thread first allocates 64K worth of memory. It then calls DosCreateThread2 four times to create 4 child threads. Each child thread has 16K of stack space. Finally, the main thread sets the termination flag to allow all child threads to terminate.

Compile this example with MULTITHREAD LIBRARIES. If you are using CSet/2 or VisualAge(R) C/C++, use the /Gm+ switch.

```
#define INCL_DOSMEMMGR
#define INCL_DOSPROCESS
#define INCL_DOSERRORS

#include os2.h>
#include stdio.h>
#include stdlib.h>

#define _64K 64*1024
#define _48K 48*1024
#define _32K 32*1024
#define _16K 16*1024

void _System TestThread1(void);
void _System TestThread2(void);
void _System TestThread3(void);
void _System TestThread4(void);

BOOL flTerminate = FALSE;

int main (VOID) {

   APIRET rc;                /* Return code */
   void *pStackBase;         /* Pointer to stack base */
   THREADCREATE tc[4]={0};  /* Thread create structures */
   int i;

   /* Allocate 64K of memory */

   rc = DosAllocMem(pStackBase, _64K, PAG_COMMIT | PAG_WRITE);

   if (rc != NO_ERROR) {
      printf("DosAllocMem failed, rc=%d\n", rc);
      return(1);
   }

   /* Set up thread structures (4 threads). */

   tc[0].cbSize   = sizeof(THREADCREATE);
   tc[0].pfnStart = (PFNTHREAD)TestThread1
   tc[0].pStack   = (PBYTE)pStackBase + _64K;    /* Top of stack (not bottom) */
   tc[0].lFlag    = CREATE_READY | STACK_SPARSE;
   tc[0].cbStack  = _16K;                        /* Each thread has 16K stack */

   tc[1].cbSize   = sizeof(THREADCREATE);
   tc[1].pfnStart = (PFNTHREAD)TestThread2
   tc[1].pStack   = (PBYTE)pStackBase + _48K;    /* Top of stack (not bottom) */
   tc[1].lFlag    = CREATE_READY | STACK_SPARSE;
   tc[1].cbStack  = _16K;                        /* Each thread has 16K stack */

   tc[2].cbSize   = sizeof(THREADCREATE);
   tc[2].pfnStart = (PFNTHREAD)TestThread3
   tc[2].pStack   = (PBYTE)pStackBase + _32K;    /* Top of stack (not bottom) */
   tc[2].lFlag    = CREATE_READY | STACK_SPARSE;
   tc[2].cbStack  = _16K;                        /* Each thread has 16K stack */

   tc[3].cbSize   = sizeof(THREADCREATE);
   tc[3].pfnStart = (PFNTHREAD)TestThread4
   tc[3].pStack   = (PBYTE)pStackBase + _16K;    /* Top of stack (not bottom) */
   tc[3].lFlag    = CREATE_READY | STACK_SPARSE;
   tc[3].cbStack  = _16K;                        /* Each thread has 16K stack */

   /* Create 4 child threads. */

   for (i=0; i4; i++) {
      rc = DosCreateThread2(tc[i]);
      if (rc != NO_ERROR) {
        printf( "DosCreateThread2 failed, rc = %d\n", rc);
        return(1);
      } else
           printf("DosCreateThread2 was successful, tid=%d\n", tc[i].pTid);
   }

   flTerminate = TRUE;

   /* Wait for all child threads to terminate. */
   for (i=0; i4; i++) {
      DosWaitThread((tc[i].pTid), DCWW_WAIT);
   }

   DosFreeMem(pStackBase);
   return(0);
}

void _System TestThread1(void)
{
   APIRET rc;
   PTIB   ptib;
   PPIB   ppib;

   rc = DosGetInfoBlocks(ptib, ppib);
   if (rc != NO_ERROR) {
      printf("TestThread1 DosGetInfoBlocks failed rc = %d\n", rc);
      return;
   }

   printf("TestThread1 base of stack at 0x%08X, top of stack at 0x%08X\n",
             ptib->tib_pstack, ptib->tib_pstacklimit);

   while (flTerminate == FALSE) {
      DosSleep(1000);
   }
}

void _System TestThread2(void)
{
   APIRET rc;
   PTIB   ptib;
   PPIB   ppib;

   rc = DosGetInfoBlocks(ptib, ppib);
   if (rc != NO_ERROR) {
      printf("TestThread2 DosGetInfoBlocks failed rc = %d\n", rc);
      return;
   }

   printf("TestThread2 base of stack at 0x%08X, top of stack at 0x%08X\n",
             ptib->tib_pstack, ptib->tib_pstacklimit);

   while (flTerminate == FALSE) {
      DosSleep(1000);
   }
}

void _System TestThread3(void)
{
   APIRET rc;
   PTIB   ptib;
   PPIB   ppib;

   rc = DosGetInfoBlocks(ptib, ppib);
   if (rc != NO_ERROR) {
      printf("TestThread3 DosGetInfoBlocks failed rc = %d\n", rc);
      return;
   }
   printf("TestThread3 base of stack at 0x%08X, top of stack at 0x%08X\n",
             ptib->tib_pstack, ptib->tib_pstacklimit);

   while (flTerminate == FALSE) {
      DosSleep(1000);
   }
}

void _System TestThread4(void)
{
   APIRET rc;
   PTIB   ptib;
   PPIB   ppib;

   rc = DosGetInfoBlocks(ptib, ppib);
   if (rc != NO_ERROR) {
      printf("TestThread4 DosGetInfoBlocks failed rc = %d\n", rc);
      return;
   }

   printf("TestThread4 base of stack at 0x%08X, top of stack at 0x%08X\n",
             ptib->tib_pstack, ptib->tib_pstacklimit);

   while (flTerminate == FALSE) {
      DosSleep(1000);
   }
}
```

DosDumpProcess

Purpose

DosDumpProcess initiates a process dump from a specified process. This may be used as part of an error handling routine to gather information about an error that may be analyzed later using the OS/2 System Dump Formatter. Configuration of Process Dump may be done using the PDUMPSYS, PDUMPUSR, and PROCDUMP commands.

Syntax

```
#define INCL_DOSMISC
#include os2.h>
```

APIRET APIENTRY DosDumpProcess **(ULONG Flag, ULONG Drive, PID Pid)**

Parameters

flag (ULONG)   input

Flags specify the function to be performed

DDP_DISABLEPROCDUMP 0x00000000L Disable process dumps.

DDP_ENABLEPROCDUMP 0x00000001L Enable process dumps.

DDP_PERFORMPROCDUMP 0x00000002L Perform process dump.

drive (ULONG)   input

The ASCII character for the drive on which process dump files are to be created. This is required only with the DDP_ENABLEPROCDUMP.

Note: Use the PROCDUMP command to customize fully the drive and path.

pid (PID)   input

The process to be dumped. 0L specified the current process; otherwise a valid process ID must be specified.

Note: Use the PDUMPUSR command to specify what information will be dumped. Use the PROCDUMP command to customize options per process and in particular to specify whether child or parent process will be dumped. This parameter is actioned only with DDP_PERFORMPROCDUMP.

Returns

ulrc (APIRET)   returns

Return Code.

DosDumpProcess returns the following value

87	ERROR_INVLAID PARAMETER

Remarks

For maximum flexibility the use of DosDumpProcess should be limited to the DDP_PERFORMPROCDUMP function. This allows you to specify whether Process Dump should be enabled through the use of the PROCDUMP command. You may customize Process Dump completely through use of the PDUMPUSR, PDUMPSYS, AND PROCDUMP commands. For further information, see PROCDUMP.DOC in the OS2\SYSTEM\RAS directory. DDP_ENABLEPROCDUMP and DDP_DISABLEPROCDUMP are provided for backwards compatibility only.

Related Functions

- DosForceSystemDump
- DosSysTrace

Example Code

```
int main (int argc, char *argv[], char *envp[])
{
   APIRET rc;

   /* Take a process dump;leave drive specification as specified by the user in the   */
   /* PROCDUMP command. If the user has not enabled process dump using PROCDUMP ON, then  */
   /* ERROR_INVALID_PARAMETER is returned.   */
   rc=DosDumpProcess(DDP_PERFORMPROCDUMP,0L,0L);
   if (rc!=0) {
      printf("DosDumpProcess returned%u\n",rc);
      return rc;
   } /* endif */

   return 0;
}
```

DosFindFirst

Purpose

DosFindFirst finds the first file object or group of file objects whose names match the specification. The specification can include extended attributes (EA) associated with a file or directory.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosFindFirst **(PSZ pszFileSpec, PHDIR phdir, ULONG flAttribute, PVOID pfindbuf, ULONG cbBuf, PULONG pcFileNames, ULONG ulInfoLevel)**

Parameters

pszFileSpec PSZ)   input

Address of the ASCIIZ path name of the file or subdirectory to be found.

The name component can contain global file name characters.

phdir PHDIR)   in/out

Address of the handle associated with this DosFindFirst request.

The values that can be specified for the handle are

HDIR_SYSTEM (0x00000001)
The system assigns the handle for standard output, which is always available to a process.

HDIR_CREATE (0xFFFFFFFF)
The system allocates and returns a handle. Upon return to the caller, *phdir* contains the handle allocated by the system.

The DosFindFirst handle is used with subsequent DosFindNext requests. Reuse of this handle in another DosFindFirst request closes the association with the previous DosFindFirst request, and opens a new association with the current DosFindFirst request.

flAttribute ULONG) input
Attribute value that determines the file objects to be searched for.

The bit values are shown in the following list

Bits	Description
31 14	Reserved; must be 0.
13	MUST_HAVE_ARCHIVED (0x00002000) Must-Have Archive bit; excludes files without the archive bit set if bit 13 is set to 1. Files may have the Archive bit set if bit 13 is set to 0.
12	MUST_HAVE_DIRECTORY (0x00001000) Must-Have Subdirectory bit; excludes files that are not subdirectories if bit 12 is set to 1. Files may have the Subdirectory bit set if bit 12 is set to 0.
11	Reserved; must be 0.
10	MUST_HAVE_SYSTEM (0x00000400) Must-Have System File bit; excludes nonsystem files if bit 10 is set to 1. Files may be system files if bit 10 is set to 0.
9	MUST_HAVE_HIDDEN (0x00000200) Must-Have Hidden File bit; excludes nonhidden files if bit 9 is set to 1. Files may be nonhidden if bit 9 is set to 0.
8	MUST_HAVE_READONLY (0x00000100) Must-Have Read-Only File bit; excludes writeable files if bit 8 is set to 1. Files may be read-only if bit 8 is set to 0.
7 6	Reserved; must be 0.
5	FILE_ARCHIVED (0x00000020) May-Have Archive bit; includes files with the Archive bit set if bit 5 is set to 1. Excludes files with the Archive bit set if bit 5 is set to 0.
4	FILE_DIRECTORY (0x00000010) May-Have Subdirectory bit; includes files that are subdirectories if bit 4 is set to 1. Excludes files that are subdirectories if bit 4 is set to 0.
3	Reserved; must be 0.
2	FILE_SYSTEM (0x00000004) May-Have System File bit; includes system files if bit 2 is set to 1. Excludes system files if bit 2 is set to 0.
1	FILE_HIDDEN (0x00000002) May-Have Hidden File bit; includes hidden files if bit 1 is set to 1. Excludes hidden files if bit 1 is set to 0.
0	FILE_READONLY (0x00000001) May-Have Read-Only File bit; includes read only files if bit 0 is set to 1. Excludes read only files if bit 0 is set to 0.

These bits may be set individually or in combination. For example, an attribute value of 0x00000021 (bits 5 and 0 set to 1) indicates searching for read-only files that have been archived.

Bits 8 through 13 are Must-Have flags. These allow you to obtain files that definitely have the given attributes. For example, if the Must-Have Subdirectory bit is set to 1, then all returned items are subdirectories.

If a Must-Have bit is set to 1, and the corresponding May-Have bit is set to 0, no items are returned for that attribute.

The attribute FILE_NORMAL (0x00000000) can be used to include files with any of the above bits set.

flAttribute cannot specify the volume label. Volume labels are queried using DosQueryFSInfo.

pfindbuf PVOID) in/out

Result buffer.

The result buffer from DosFindFirst should be less than 64 KB.

Address of the directory search structures for file object information Levels 1 through 3 and 13. The structure required for *pfindbuf* is dependent on the value specified for *ulInfoLevel*. The information returned reflects the most recent call to DosClose or DosResetBuffer.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)

On output, *pfindbuf* contains the FILEFINDBUF3 data structure without the last two fields *cchName* and *achName*. This is used without EAs.

The *oNextEntryOffset* field indicates the number of bytes from the beginning of the current structure to the beginning of the next structure. When this field is 0, the last structure has been reached.

Level 11 File Information (*ulInfoLevel* == FIL_STANDARDL)

pInfo contains the FILESTATUS3L data structure, to which file information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)

On output, *pfindbuf* contains the FILEFINDBUF4 data structure without the last two fields *cchName* and *achName*. This is used with EAs.

The *cbList* field contains the size, in bytes, of the file s entire EA set on disk. You can use this field to calculate the maximum size of the buffer needed for Level 3 file information. The size of the buffer required to hold the entire EA set is less than or equal to twice the size of the EA set on disk.

Level 12 File Information (*ulInfoLevel* == FIL_QUERYEASIZEL)

pInfo contains the FILESTATUS4L data structure. This is similar to the Level 11 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is a ULONG. On output, this field contains the size, in bytes, of the file s entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel*. The buffer size is less than or equal to twice the size of the file s entire EA set on disk.

Level 3 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLIST)

On input, *pfindbuf* contains an EAOP2 data structure. *fpGEA2List* contains a pointer to a GEA2 list, which defines the attribute names whose values are to be returned. Entries in the GEA2 list must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry.

On output, *pfindbuf* contains a structure with a set of records, each aligned on a doubleword boundary. These records represent the directory entry and associated EAs for the matched file object. *pfindbuf* has the following format

- The EAOP2 data structure, with the *fpFEA2List* pointer incorrect.

  The EAOP2 data structure occurs only once in the *pfindbuf* buffer. The rest of these records are repeated for the remainder of the file objects found.

- A FILEFINDBUF3 data structure without the last two fields *cchName* and *achName*.

- A FEA2LIST data structure contained in and related to the FILEFINDBUF3 returned.

- Length of the name string of the file object (*cbName*)

- Name of the file object matched by the input pattern (*achName*)

Even if there is not enough room to hold all of the requested information, as for return code ERROR_BUFFER_OVERFLOW, the *cbList* field of the FEA2LIST data structure is valid if there is at least enough space to hold it.

When buffer overflow occurs, *cbList* contains the size on disk of the entire EA set for the file, even if only a subset of its attributes was requested. The size of the buffer required to hold the EA set is less than or equal to twice the size of the EA set on disk. If no error occurs, *cbList* includes the pad bytes (for doubleword alignment) between FEA2 structures in the list.

If a particular attribute is not attached to the object, *pfindbuf* has an FEA2 structure containing the name of the attribute, and the length value is 0.

The GEA2 list contained inside *pfindbuf* during a Level 3 DosFindFirst and DosFindNext call is not read-only ; it is used by the operating system. When the function returns, the list is restored to its original state, but inside the function, the list is manipulated by the operating system. This is of concern to a multithreaded application, where two different threads might use the same GEA2 list as input. If one thread calls DosFindFirst or DosFindNext while another thread is inside DosFindFirst or DosFindNext, the second thread will fail with a return code of ERROR_BAD_FORMAT.

For Level 13 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLISTL)

On input, *pfindbuf* contains an EAOP2 data structure. *fpGEA2List* contains a pointer to a GEA2 list, which defines the attribute names whose values are to be returned. Entries in the GEA2 list must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry.

On output, *pfindbuf* contains a structure with a set of records, each aligned on a doubleword boundary. These records represent the directory entry and associated EAs for the matched file object. *pfindbuf* has the following format

- The EAOP2 data structure, with the *fpFEA2List* pointer incorrect.

  The EAOP2 data structure occurs only once in the *pfindbuf* buffer. The rest of these records are repeated for the remainder of the file objects found.

- A FILEFINDBUF3L data structure without the last two fields *cchName* and *achName* .

- A FEA2LIST data structure contained in and related to the FILEFINDBUF3L returned.

- Length of the name string of the file object (*cbName*)

- Name of the file object matched by the input pattern (*achName*)

Even if there is not enough room to hold all of the requested information, as for return code ERROR_BUFFER_OVERFLOW, the *cbList* field of the FEA2LIST data structure is valid if there is at least enough space to hold it.

When buffer overflow occurs, *cbList* contains the size on disk of the entire EA set for the file, even if only a subset of its attributes was requested. The size of the buffer required to hold the EA set is less than or equal to twice the size of the EA set on disk. If no error occurs, *cbList* includes the pad bytes (for doubleword alignment) between FEA2 structures in the list.

If a particular attribute is not attached to the object, *pfindbuf* has an FEA2 structure containing the name of the attribute, and the length value is 0.

The GEA2 list contained inside *pfindbuf* during a Level 13 DosFindFirst and DosFindNext call is not read-only ; it is used by the operating system. When the function returns, the list is restored to its original state, but inside the function, the list is manipulated by the operating system. This is of concern to a multithreaded application, where two different threads might use the same GEA2 list as input. If one thread calls DosFindFirst or DosFindNext while another thread is inside DosFindFirst or DosFindNext, the second thread will fail with a return code of ERROR_BAD_FORMAT.

cbBuf ULONG)   input

The length, in bytes, of *pfindbuf* .

pcFileNames PULONG)   in/out

Pointer to the number of entries

Input	The address of the number of matching entries requested in *pfindbuf* .
Output	The number of entries placed into *pfindbuf* .

ulInfoLevel ULONG)   input

The level of file information required.

Possible values are

1	FIL_STANDARD Level 1 file information (return standard file information).
11	FIL_STANDARDL Level 11 file information

2	FIL_QUERYEASIZE Level 2 file information
12	FIL_QUERYEASIZEL Level 12 file information
3	FIL_QUERYEASFROMLIST Level 3 file information (return requested EA).
13	FIL_QUERYEASFROMLISTL Level 13 file information (return requested EA).

The structures described in *pfindbuf* indicate the information returned for each of these levels.

Regardless of the level specified, a DosFindFirst request (and an associated DosFindNext request on a handle returned by DosFindFirst) always includes Level 1 information as part of the information that is returned; however, when Level 1 information is specifically requested, and *flAttribute* specifies hidden files, system files, or subdirectory files, an inclusive search is made. That is, all normal file entries plus all entries matching any specified attributes are returned. Normal files are files without any mode bits set. They may be read from or written to.

Returns

ulrc APIRET) returns

Return Code.

DosFindFirst returns one of the following values

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
6	ERROR_INVALID_HANDLE
18	ERROR_NO_MORE_FILES
26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
111	ERROR_BUFFER_OVERFLOW
113	ERROR_NO_MORE_SEARCH_HANDLES
206	ERROR_FILENAME_EXCED_RANGE
208	ERROR_META_EXPANSION_TOO_LONG
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT
275	ERROR_EAS_DIDNT_FIT

Remarks

The result buffer from DosFindFirst should be less than 64KB.

DosFindFirst returns directory entries (up to the number requested in *pcFileNames*) and extended-attribute information for as many files or subdirectories whose names, attributes, and EAs match the specification, and whose information fits in *pfindbuf* . On output, *pcFileNames* contains the actual number of directory entries returned.

The file name pointed to by *pszFileSpec* can contain global file-name characters.

DosFindNext uses the directory handle associated with DosFindFirst to continue the search started by the DosFindFirst request.

Any nonzero return code, except ERROR_EAS_DIDNT_FIT, indicates that no handle has been allocated. This includes such non-error return codes as ERROR_NO_MORE_FILES.

For ERROR_EAS_DIDNT_FIT, a search handle is returned, and a subsequent call to DosFindNext gets the next matching entry in the directory. You can use DosQueryPathInfo to retrieve the EAs for the matching entry by using the same EA arguments used for the DosFindFirst call, and the name that was returned by DosFindFirst.

For ERROR_EAS_DIDNT_FIT, only information for the first matching entry is returned. This entry is the one whose extended attributes did not fit in the buffer. The information returned is in the format of that returned for information Level 2. No further entries are returned in the buffer, even if they could fit in the remaining space.

The GEA2 list contained inside *pfindbuf* during a Level 3 DosFindFirst and DosFindNext call is not read-only , it is used by the operating system. When the function returns, the list is restored to its original state, but inside the function, the list is manipulated by the operating system. This is of concern to a multithreaded application, where two different threads might use the same GEA2 list as input. If one thread calls DosFindFirst or DosFindNext while another thread is inside DosFindFirst or DosFindNext, the second thread will fail with a return code of ERROR_BAD_FORMAT.

Related Functions

- DosClose
- DosFindClose
- DosFindNext
- DosQueryFileInfo
- DosQueryPathInfo
- DosQuerySysInfo
- DosResetBuffer
- DosSearchPath
- DosSetFileInfo
- DosSetPathInfo

Example Code

This example lists all the normal files that are in the directory from where the example is invoked.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values */
#include os2.h
#include stdio.h

int main (VOID)
HDIR          hdirFindHandle = HDIR_CREATE;
FILEFINDBUF3L FindBuffer     = 0;      /* Returned from FindFirst/Next */
ULONG         ulResultBufLen = sizeof(FILEFINDBUF3L);
ULONG         ulFindCount    = 1;        /* Look for 1 file at a time    */
APIRET        rc             = NO_ERROR; /* Return code                  */

rc = DosFindFirst( "*.*",                /* File pattern - all files     */
hdirFindHandle,      /* Directory search handle      */
FILE_NORMAL,          /* Search attribute             */
FindBuffer,          /* Result buffer                */
ulResultBufLen,       /* Result buffer length         */
ulFindCount,         /* Number of entries to find    */
FIL_STANDARDL);       /* Return Level 11 file info     */

if (rc != NO_ERROR)
printf("DosFindFirst error return code = %u\n",rc);
return 1;
 else
printf ("%s\n", FindBuffer.achName);   /* Print file name             */
 /* endif */

/* Keep finding the next file until there are no more files */
while (rc != ERROR_NO_MORE_FILES)
ulFindCount = 1;                      /* Reset find count.            */

rc = DosFindNext(hdirFindHandle,      /* Directory handle             */
FindBuffer,         /* Result buffer                */
ulResultBufLen,      /* Result buffer length         */
ulFindCount);       /* Number of entries to find    */

if (rc != NO_ERROR  rc != ERROR_NO_MORE_FILES)
printf("DosFindNext error return code = %u\n",rc);
return 1;
 else
printf ("%s\n", FindBuffer.achName);    /* Print file name */

 /* endwhile */
```

```
rc = DosFindClose(hdirFindHandle);    /* Close our directory handle */
if (rc != NO_ERROR)
printf("DosFindClose error return code = %u\n",rc);
return 1;

return NO_ERROR;
```

DosFindNext

Purpose

DosFindNext finds the next set of file objects whose names match the specification in a previous call to DosFindFirst or DosFindNext.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosFindNext **(HDIR hDir, PVOID pfindbuf, ULONG cbfindbuf, PULONG pcFilenames)**

Parameters

hDir HDIR) input

The handle of the directory.

pfindbuf PVOID) in/out

The address of the directory search information structure.

The information returned reflects the most recent call to DosClose or DosResetBuffer.

For the continuation of a Level 3 (FIL_QUERYEASFROMLIST) File Information search, this buffer should contain input in the same format as a Level 3 File Information search by DosFindFirst.

See the description of the *pfindbuf* parameter in DosFindFirst for information about the output data that the file system driver places into this buffer.

cbfindbuf ULONG) input

The length, in bytes, of *pfindbuf* .

pcFilenames PULONG) in/out

Pointer to the number of entries.

Input	The address of the number of matching entries requested in *pfindbuf* .
Output	The number of entries placed into *pfindbuf* .

Returns

ulrc APIRET) returns

Return Code.

DosFindNext returns one of the following values

0	NO_ERROR
6	ERROR_INVALID_HANDLE
18	ERROR_NO_MORE_FILES
26	ERROR_NOT_DOS_DISK
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW

Remarks

If ERROR_BUFFER_OVERFLOW is returned, further calls to DosFindNext start the search from the same entry.

If ERROR_EAS_DIDNT_FIT is returned, the buffer is too small to hold the extended attributes (EAs) for the first matching entry being returned. A subsequent call to DosFindNext gets the next matching entry. This enables the search to continue if the extended attributes being returned are too large for the buffer. You can use DosQueryPathInfo to retrieve the extended attributes for the matching entry by using the same EA arguments used for the call to DosFindFirst, and the name that was returned by DosFindFirst,

In the case of ERROR_EAS_DIDNT_FIT, only information for the first matching entry is returned. This is the entry whose extended attributes did not fit in the buffer. The information returned is in the format of Level 2 or Level 12 (FIL_QUERYEASIZE) File Information (FILEFINDBUF4 or FILEFINDBUF4L). No further entries are returned in the buffer, even if they could fit in the remaining space.

Related Functions

- DosClose
- DosFindClose
- DosFindFirst
- DosQueryFileInfo
- DosQueryPathInfo
- DosQuerySysInfo
- DosResetBuffer
- DosSearchPath
- DosSetFileInfo
- DosSetPathInfo

Example Code

This example lists all the normal files that are in the directory from where the example is invoked.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values */
#include os2.h
#include stdio.h

int main (VOID)
HDIR          hdirFindHandle = HDIR_CREATE;
FILEFINDBUF3L  FindBuffer     = 0;      /* Returned from FindFirst/Next */
ULONG         ulResultBufLen = sizeof(FILEFINDBUF3L);
ULONG         ulFindCount    = 1;        /* Look for 1 file at a time    */
APIRET        rc             = NO_ERROR; /* Return code                  */

rc = DosFindFirst( "*.*",                /* File pattern - all files     */
hdirFindHandle,      /* Directory search handle      */
FILE_NORMAL,          /* Search attribute             */
FindBuffer,          /* Result buffer                */
ulResultBufLen,       /* Result buffer length         */
ulFindCount,         /* Number of entries to find    */
FIL_STANDARDL);       /* Return level 1 file info     */

if (rc != NO_ERROR)
printf("DosFindFirst error return code = %u\n",rc);
return 1;
 else
printf ("%s\n", FindBuffer.achName);   /* Print file name             */
 /* endif */

/* Keep finding the next file until there are no more files */
while (rc != ERROR_NO_MORE_FILES)
ulFindCount = 1;                      /* Reset find count.            */

rc = DosFindNext(hdirFindHandle,      /* Directory handle             */
FindBuffer,         /* Result buffer                */
ulResultBufLen,      /* Result buffer length         */
ulFindCount);       /* Number of entries to find    */

if (rc != NO_ERROR  rc != ERROR_NO_MORE_FILES)
printf("DosFindNext error return code = %u\n",rc);
```

```
return 1;
 else
printf ("%s\n", FindBuffer.achName);    /* Print file name */

 /* endwhile */

rc = DosFindClose(hdirFindHandle);    /* Close our directory handle */
if (rc != NO_ERROR)
printf("DosFindClose error return code = %u\n",rc);
return 1;

return NO_ERROR;
```

DosForceSystemDump

Purpose

DosForceSystemDump initiates a stand-alone dump. The system terminates abruptly without shutdown as soon as the dump is inititated.

Syntax

```
#define INCL_DOSMISC
#include os2.h>
```

APIRET APIENTRY DosForceSystemDump **(ULONG reserved)**

Parameters

reserved(ULONG)   input

Returns

ulrc (APIRET)   returns

Return Code.

DosForceSystemDump returns the following value

87	ERROR_INVALID_PARAMETER

Related Functions

- DosDumpProcess
- DosSysTrace

Example Code

```
int main(int argc, char *argv[], char *envp[])
{
   APIRET rc;

   rc=DosForceSystemDump(0L);   /* will kill the system with a system dump */
                                /* does not return unless an error occurs */

   printf("DosForceSystemDump returned %u\n",rc);

   return rc;
}
```

DosGetProcessorStatus

Purpose

DosGetProcessorStatus returns the ONLINE or OFFLINE status of each processor of an SMP system. The processor status may be set using DosSetProcessorStatus. ONLINE status imples the processor is available for running work. OFFLINE status implies the porcessor is not available for running work.

Syntax

```
#define INCL_DOS
#define INCL_DOSSPINLOCK
#include os2.h>
```

APIRET APRIENTRY DosGetProcessorStatus **(ULONG procid, PULONG status)**

Parameters

procid (ULONG)   input
Procesor ID numbered 1 through n, where there are n processors in total

status (PULONG)   output
Status is defined as follows

PROC_OFFLINE 0x00000000	Processor is offline
PROC_ONLINE 0x00000001	Processor is online

Returns

ulrc (APIRET)   returns
Return Code.

DosGetProcessorStatus returns one of the following values

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

Related Functions

- DosSetProcessorStatus

Example Code

```
int main(int argc, char *argv[], char *envp[])
{
   APIRET rc=0;
   ULONG procid;
   ULONG status;
   int i;

   if (argc == 1) {
      for (procid=1; rc==0 ;++procid) {
         rc = DosGetProcessorStatus(procid, status);
         if (rc==0) {
            if (status == PROC_OFFLINE) printf("Processor %u offline\n", procid);
            else printf("Processor %u online\n", procid);
         } /* endif */
      } /* endfor */

   } else for (i=1; iargc ; ++i) {
      procid = atol(argv[i]);
      rc = DosGetProcessorStatus(procid, status);
      if (rc) printf("DosGetProcesorStatus returned %u\n",rc);
      else {
         if (status == PROC_OFFLINE) printf("Processor %u offline\n", procid);
         else printf("Processor %u online\n", procid);
      } /* endif */
   } /* endfor */

   return rc;
}
```

DosListIO

Purpose

DosListIO performs the specified number of seek/read and/or seek/write operations.

Syntax

```
#define INCL_DOSFILEMGR
#include os2.h>
```

APIRET DosListIO **(ULONG CmdMode, ULONG NumEntries, PLISTIO pListIO)**

Parameters

CmdMode (ULONG) input
This specifies the mode in which the operations should be performed. Valid modes are

LISTIO_ORDERED
Operations are performed synchronously in the given order.

LISTIO_UNORDERED
Operations are performed independent of order.

NumEntries (ULONG) input
The number of seek/read or seek/write operations in the list.

pListIO (PLISTIO) input/output
Pointer to an array of *NumEntries* **LISTIO** data structures which contain the information necessary to perform the seek/read and seek/write operations.

Returns

ulrc (APIRET) returns
Return Code.

DosListIO returns one of the following values

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
19	ERROR_WRITE_PROTECT
26	ERROR_NOT_DOS_DISK
29	ERROR_WRITE_FAULT
33	ERROR_LOCK_VIOLATION
87	ERROR_INVALID_PARAMETER
109	ERROR_BROKEN_PIPE
234	ERROR_MORE_DATA

Remarks

DosListIO applies the same restrictions for each seek/read and seek/write control block as would be applied if the requests were issued separately with DosSetFilePtr, DosRead, and DosWrite.

Each request control block contains fields for the Actual number of bytes read/written and the operation return code. These fields are updated upon completion of each request, therefore care must be taken that the memory containing the control block array not be deallocated or manipulated by another thread before the DosListIO request returns.

There are two valid modes for the list of I/O operations to be processed

- Ordered - This mode guarantees the order in which the operations will be performed. The API will return with an error code corresponding to the first failed request and will leave the following requests unissued. This provide a synchronous sequence of automatic seek/read and seek/write requests. This is the only mode that is compatible with file systems other than the raw file system.

- Unordered - This mode does not guarantee the order of issue or completion of the requests. The API will return with an error code if any request fails. Additionally, each request in the list will be issued, even those following a failed operation. This mode is valid for the raw file system only.

Related Functions

- DosOpen

- DosSetFilePtr

- DosRead

- DosWrite

Example Code

The following is NOT a complete usable program. It is simply intended to provide an idea of how to use Raw I/O File System APIs (e.g. DosOpen, DosListIO, and DosClose).

This example opens physical disk #1 for reading and physical disk #2   for writing. Using DosListIO, 10 megabytes of data is transferred disk #1 to disk #2. Finally, DosClosed is issued to close the disk handles.

It is assumed that the size of each of the two disks is at least 10 megabytes.

```
#define INCL_DOSFILEMGR          /* Include File Manager APIs */
#define INCL_DOSMEMMGR           /* Includes Memory Management APIs */
#define INCL_DOSERRORS           /* DOS Error values */
#include os2.h>
#include stdio.h>
#include stdlib.h>
#include string.h>

#define SIXTY_FOUR_K 0x10000
#define ONE_MEG     0x100000
#define TEN_MEG     10*ONE_MEG

#define UNC_DISK1  "\\\\.\\Physical_Disk1"
#define UNC_DISK2  "\\\\.\\Physical_Disk2"

int main(void) {
   LISTIO listIOCtrlBlks[2];          /* List IO control blocks   */
   ULONG  ulNumCtrlBlks;              /* Number of control blocks */
   HFILE  hfDisk1        = 0;         /* Handle for disk #1 */
   HFILE  hfDisk2        = 0;         /* Handle for disk #2 */
   ULONG  ulAction       = 0;         /* Action taken by DosOpen */
   UCHAR  uchFileName1[20]  = UNC_DISK1, /* UNC Name of disk 1 */
          uchFileName2[20]  = UNC_DISK2; /* UNC Name of disk 2 */
   PBYTE  pBuffer        = 0;
   ULONG  cbTotal        = 0;

   APIRET rc = NO_ERROR;              /* Return code */

   /* Open a raw file system disk #1 for reading */
   rc = DosOpen(uchFileName1,               /* File name */
                hfDisk1,                   /* File handle */
                ulAction,                  /* Action taken by DosOpen */
                0L,                         /* no file size */
                FILE_NORMAL,                /* File attribute */
                OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
                OPEN_SHARE_DENYNONE |       /* Access mode */
                OPEN_ACCESS_READONLY,
                0L);                        /* No extented attributes */
   if (rc != NO_ERROR) {
      printf("DosOpen error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Open a raw file system disk #2 for writing */
   rc = DosOpen(uchFileName2,               /* File name */
                hfDisk2,                   /* File handle */
                ulAction,                  /* Action taken by DosOpen */
                0L,                         /* no file size */
                FILE_NORMAL,                /* File attribute */
                OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
```

```
             OPEN_SHARE_DENYNONE |       /* Access mode */
             OPEN_ACCESS_READWRITE,
             0L);                        /* No extented attributes */
if (rc != NO_ERROR) {
   printf("DosOpen error rc = %u\n", rc);
   return(1);
} /* endif */


/* Allocate 64K of memory for transfer operations */
rc = DosAllocMem((PPVOID)pBuffer, /* Pointer to buffer */
                  SIXTY_FOUR_K,      /* Buffer size */
                  PAG_COMMIT |     /* Allocation flags */
                  PAG_READ |
                  PAG_WRITE);
if (rc != NO_ERROR) {
   printf("DosAllocMem error rc = %u\n", rc);
   return(1);
} /* endif */

/* Initialize listIO control blocks */
memset(listIOCtrlBlks, 0, sizeof(listIOCtrlBlks));

listIOCtrlBlks[0].hFile = hfDisk1;       /* Handle for disk 1 */
listIOCtrlBlks[0].CmdFlag = LISTIO_READ | /* Read operation */
                            FILE_CURRENT;
listIOCtrlBlks[0].Offset = 0;
listIOCtrlBlks[0].pBuffer = (PVOID)pBuffer;
listIOCtrlBlks[0].NumBytes = SIXTY_FOUR_K;

listIOCtrlBlks[1].hFile = hfDisk2;        /* Handle for disk 2 */
listIOCtrlBlks[1].CmdFlag = LISTIO_WRITE | /* Write operation */
                            FILE_CURRENT;
listIOCtrlBlks[1].Offset = 0;
listIOCtrlBlks[1].pBuffer = (PVOID)pBuffer;
listIOCtrlBlks[1].NumBytes = SIXTY_FOUR_K;

while (cbTotal  TEN_MEG) {


   ulNumCtrlBlks = 2;
   rc = DosListIO(LISTIO_ORDERED,
                  ulNumCtrlBlks,
                  listIOCtrlBlks);
   if (rc != NO_ERROR) {
      printf("DosListIO error rc = %u\n", rc);
      break;
   } else {

      /* Check return code from the read operation */
      if (listIOCtrlBlks[0].RetCode != NO_ERROR) {
         printf("DosListIO read operation failed, rc = %u\n",
                 listIOCtrlBlks[0].RetCode);
        return 1;
      }

      /* Check return code from the write operation */
      if (listIOCtrlBlks[0].RetCode != NO_ERROR) {
         printf("DosListIO write operation failed, rc = %u\n",
                 listIOCtrlBlks[0].RetCode);
         return 1;
      }
   }

   if (listIOCtrlBlks[0].Actual != listIOCtrlBlks[1].Actual) {
      printf("Bytes read (%u) does not equal bytes written (%u)\n",
             listIOCtrlBlks[0].Actual, listIOCtrlBlks[1].Actual);
      return 1;
   }

   cbTotal += SIXTY_FOUR_K; /* Update total transferred */

} /* end while */

printf("Transfer successfully %d bytes from disk #1 to disk #2.\n",
       cbTotal);

/* Free allocated memmory */
rc = DosFreeMem(pBuffer);
if (rc != NO_ERROR) {
   printf("DosFreeMem error return code = %u\n", rc);
   return 1;
}
```

```
   rc = DosClose(hfDisk1);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
   }

   rc = DosClose(hfDisk2);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
   }
return NO_ERROR;
}
```

DosListIOL

Purpose

DosListIOL performs the specified number of seek/read or seek/write operations or both.

Syntax

```
#define INCL_DOSFILEMGR
#include os2.h>
```

APIRET DosListIOL **(LONG CmdMode, LONG NumEntries, PLISTIOL pListIO)**

Parameters

CmdMode (LONG) input

This specifies the mode in which the operations should be performed. Valid modes are

LISTIO_ORDERED

Operations are performed synchronously in the given order.

LISTIO_UNORDERED

Operations are performed independent of order.

NumEntries (LONG) input

The number of seek/read or seek/write operations in the list.

pListIOL (PLISTIO) input/output

Pointer to an array of *NumEntries* **LISTIO** data structures which contain the information necessary to perform the seek/read and seek/write operations.

Returns

ulrc (APIRET) returns

Return Code.

DosListIOL returns one of the following values

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
19	ERROR_WRITE_PROTECT
26	ERROR_NOT_DOS_DISK
29	ERROR_WRITE_FAULT
33	ERROR_LOCK_VIOLATION

87	ERROR_INVALID_PARAMETER
109	ERROR_BROKEN_PIPE
234	ERROR_MORE_DATA

Remarks

DosListIOL applies the same restrictions for each seek/read and seek/write control block as would be applied if the requests were issued separately with DosSetFilePtrL, DosRead, and DosWrite.

Each request control block contains fields for the Actual number of bytes read/written and the operation return code. These fields are updated upon completion of each request; therefore, care must be taken that the memory containing the control block array not be deallocated or manipulated by another thread before the DosListIOL request returns.

There are two valid modes for the list of I/O operations to be processed

- Ordered - This mode guarantees the order in which the operations will be performed. The API will return with an error code corresponding to the first failed request and will leave the following requests unissued. This provides a synchronous sequence of automatic seek/read and seek/write requests. This is the only mode that is compatible with file systems other than the raw file system.
- Unordered - This mode does not guarantee the order of issue or completion of the requests. The API will return with an error code if any request fails. Additionally, each request in the list will be issued, even those following a failed operation. This mode is valid for the raw file system only.

Related Functions

- DosOpenL
- DosSetFilePtrL
- DosRead
- DosWrite

Example Code

In this example, the source file SOURCE.DAT is copied to TARGET.DAT. First, the information about the source file is obtained by calling DosQueryPathInfo. Next, the target file is created with the same size as the source file. Using a series of calls to DosListIO, the content of the source file is copied to the target file.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#define INCL_LONGLONG
#include #define SOURCE_PATHNAME "source.dat"
#define TARGET_PATHNAME "target.dat"
#define BUFFER_SIZE 4096

int main(void) {
   FILESTATUS3L  fsSource = { {0} };          /* Buffer for information about source file  */
   LONGLONG llSize;                           /* Source file size (totalcopy size) */
   HFILE  hfSource   = 0L;                    /* Handle for source file */
   HFILE  hfTarget   = 0L;                    /* Handle for target file */
   ULONG  ulAction = 0;                       /* Action taken by DosOpen*/
   LISTIOL listIOCtrlBlks[2];                 /* List IO control blocks */
   ULONG  ulNumCtrlBlks;                      /* Number of control blocks*/
   BYTE   pData[BUFFER_SIZE];                 /* Buffer to hold copy data */
   ULONG  cbData;                             /* Size of data for each IO operation */
   APIRET rc = NO_ERROR;                      /* Return code */

   /* Query information about the source file to obtain its size */
   rc = DosQueryPathInfo(SOURCE_PATHNAME, FIL_STANDARDL, fsSource, sizeof(fsSource));
   if (rc != NO_ERROR)
   {
      printf("DosQueryPathInfo failed, return code = %u\n", rc);
      return 1;
   }

   llSize = fsSource.cbFile;

   /* Open the source file for reading */
   rc = DosOpenL(SOURCE_PATHNAME,              /* File path name */
                hfSource,                     /* File handle */
                ulAction,                     /* Action taken */
                0,                             /* File primary allocation */
                FILE_ARCHIVED | FILE_NORMAL,   /* File attribute */
                OPEN_ACTION_FAIL_IF_NEW |      /* Open existing file */
                OPEN_ACTION_OPEN_IF_EXISTS,
                OPEN_FLAGS_NOINHERIT |
                OPEN_SHARE_DENYNONE  |
                OPEN_ACCESS_READWRITE,          /* Open mode of the file */
                0L);                            /* No extended attribute */

   if (rc != NO_ERROR)
   {
      printf("DosOpenL failed to open %s, rc = %u\n", SOURCE_PATHNAME, rc);
      return 1;
   }

   /* Open the target file for writing */
   rc = DosOpenL(TARGET_PATHNAME,               /* File path name */
                hfTarget,                      /* File handle */
                ulAction,                      /* Action taken */
                llSize,                         /* Target equals source file size */
                FILE_ARCHIVED | FILE_NORMAL,    /* File attribute */
                OPEN_ACTION_CREATE_IF_NEW |     /* Open new file */
                OPEN_ACTION_FAIL_IF_EXISTS,
                OPEN_FLAGS_NOINHERIT |
                OPEN_SHARE_DENYNONE  |
                OPEN_ACCESS_READWRITE,          /* Open mode of the file */
                0L);                            /* No extended attribute */

   if (rc != NO_ERROR)
   {
      printf("DosOpenL failed to create %s, rc = %u\n", TARGET_PATHNAME, rc);
      DosClose(hfSource);  /* Remember to close source file before exiting */
      return 1;
   }
```
In this example, the source file "SOURCE.DAT" is copied to "TARGET.DAT." First, the information about the source file is obtained by calling DosQueryPathInfo. Next, the target file is created with the same size as the source file. Using a series of calls to DosListIO, the content of the source file is copied to the target file.   /* Initialize listIOL control blocks */
```
   memset(listIOCtrlBlks, 0, sizeof(listIOCtrlBlks));

   listIOCtrlBlks[0].hFile = hfSource;                      /* Source file handle */
   listIOCtrlBlks[0].CmdFlag = LISTIO_READ | FILE_CURRENT;  /* Read operation */
   listIOCtrlBlks[0].Offset = 0;
   listIOCtrlBlks[0].pBuffer = (PVOID)pData;

   listIOCtrlBlks[1].hFile = hfTarget;                      /* Target file handle */
   listIOCtrlBlks[1].CmdFlag = LISTIO_WRITE | FILE_CURRENT; /* Write operation */
   listIOCtrlBlks[1].Offset = 0;
   listIOCtrlBlks[1].pBuffer = (PVOID)pData;

   while (llSize) {

      if (llSize  BUFFER_SIZE) {
         cbData = llSize;
      } else {
         cbData = BUFFER_SIZE;
      }
      llSize = llSize - cbData;   /* adjust remaining copy size */

      listIOCtrlBlks[0].NumBytes = cbData;
      listIOCtrlBlks[1].NumBytes = cbData;

      ulNumCtrlBlks = 2;
      rc = DosListIOL(LISTIO_ORDERED,
                     ulNumCtrlBlks,
                     listIOCtrlBlks);
      if (rc != NO_ERROR)
      {
         printf("DosListIOL error rc = %u\n", rc);
         break;
      }
      else
      {

         /* Check return code from the read operation */
         if (listIOCtrlBlks[0].RetCode != NO_ERROR)
         {
            printf("DosListIOL read operation failed, rc = %u\n", listIOCtrlBlks[0].RetCode);
            break;
         }

         /* Check return code from the write operation */
         if (listIOCtrlBlks[0].RetCode != NO_ERROR)
         {
            printf("DosListIOL write operation failed, rc = %u\n", listIOCtrlBlks[0].RetCode);
            break;
         }
      }
   } /* end while */

   DosClose(hfSource);                /* Close source file */
   DosClose(hfTarget);                /* Close target file */

   return NO_ERROR;
}
```

DosOpen

Purpose

DosOpen opens a physical or logical disk and returns a handle to be used to perform operations upon the disk specified.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosOpen **(PSZ pszFileName, PHFILE pHf, PULONG pulAction, ULONG cbFile, ULONG ulAttribute, ULONG fsOpenFlags, ULONG fsOpenMode, PEAOP2 peaop2)**

Parameters

pszFileName PSZ) input
Address of the ASCIIZ path name of the logical partition or physical disk to be opened.

pHf PHFILE) output
Address of the handle for the disk.

pulAction PULONG) output
Address of the variable that receives the value that specifies the action taken by the DosOpen function.

If DosOpen fails, this value has no meaning. Otherwise, the raw file system should always reutn FILE_EXISTED (1).

cbFile ULONG) input
Not used by the raw file system.

ulAttribute ULONG) input
File attributes are ignored because disks are not created.

fsOpenFlags ULONG) input
Not used by the raw file system.

fsOpenMode ULONG) input
OPEN_ACCESS_READWRITE - Only valid access mode.

OPEN_SHARE_DENYNONE - Allows other processes to read/write from disk.

OPEN_SHARE_DENYREADWRITE - Locks disk from access by other processes and file systems.

Invalid flags are OPEN_ACCESS_WRITEONLY, OPEN_ACCESS_READONLY, OPEN_SHARE_DENYREAD, OPEN_SHARE_DENYWRITE, and OPEN_FLAGS_DASD.

Valid but unimplemented flags are OPEN_FLAGS_NOINHERIT, OPEN_FLAGS_RANDOMSEQUENTIAL, OPEN_FLAGS_RANDOM, OPEN_FLAGS_SEQUENTIAL, OPEN_FLAGS_NO_LOCALITY, OPEN_FLAGS_NO_CACHE, OPEN_FLAGS_FAIL_ON_ERROR, and OPEN_FLAGS_WRITE_THROUGH.

peaop2 PEAOP2) in/out
Unused by raw file system.

Returns

ulrc APIRET) returns
Return Code.

DosOpen returns one of the following values

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED
12	ERROR_INVALID_ACCESS
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
82	ERROR_CANNOT_MAKE
87	ERROR_INVALID_PARAMETER
99	ERROR_DEVICE_IN_USE
108	ERROR_DRIVE_LOCKED
110	ERROR_OPEN_FAILED
112	ERROR_DISK_FULL
206	ERROR_FILENAME_EXCED_RANGE
231	ERROR_PIPE_BUSY

Remarks

A successful DosOpen request returns a handle for accessing the disk. The read/write pointer is set at the first byte of the disk. The position of the pointer can be changed with DosSetFilePtr or by read and write operations on the disk.

The direct open bit (OPEN_FLAGS_DASD) is not used with the raw file system. However, when using the raw file system to access logical partitions and disk locking is required, the following logic should be used. First, the application should lock the disk by passing the handle to DosDevIOCtl, Category 8, DSK_LOCKDRIVE. Second, the application should perform the desired operations on the disk. Lastly, the application should unlock the disk using DosDevIOCtl Category 8, DSK_UNLOCKDRIVE.

If locking is desired when using the raw file system on physical disk, the OPEN_SHARE_DENYREADWRITE flag should be used. The disk will automatically be unlocked when the disk is closed with DosClose.

Related Functions

- DosClose
- DosDevIOCtl

Example Code

The following is NOT a complete usable program. It is simply intended   to provide an idea of how to use Raw I/O File System APIs (e.g. DosOpen, DosRead, DosWrite, DosSetFilePtr, and DosClose).

This example opens physical disk #1 for reading and physical disk #2   for writing. DosSetFilePtr is used to set the pointer to the beginning of the disks. Using DosRead and DosWrite, 10 megabytes of data is transferred from disk #1 to disk #2. Finally, DosClosed is issued to close the disk handles.

It is assumed that the size of each of the two disks is at least 10 megabytes.

```
#define INCL_DOSFILEMGR          /* Include File Manager APIs */
#define INCL_DOSMEMMGR           /* Includes Memory Management APIs */
#define INCL_DOSERRORS           /* DOS Error values */
#include os2.h>
#include stdio.h>
#include string.h>
#define SIXTY_FOUR_K 0x10000
#define ONE_MEG     0x100000
#define TEN_MEG     10*ONE_MEG
```

```
#define UNC_DISK1  "\\\\.\\Physical_Disk1"
#define UNC_DISK2  "\\\\.\\Physical_Disk2"

int main(void) {
   HFILE  hfDisk1        = 0;      /* Handle for disk #1 */
   HFILE  hfDisk2        = 0;      /* Handle for disk #2 */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   ULONG  cbRead         = 0;      /* Bytes to read */
   ULONG  cbActualRead   = 0;      /* Bytes read by DosRead */
   ULONG  cbWrite        = 0;      /* Bytes to write */
   ULONG  ulLocation     = 0;
   ULONG  cbActualWrote  = 0;      /* Bytes written by DosWrite */
   UCHAR  uchFileName1[20]  = UNC_DISK1, /* UNC Name of disk 1 */
          uchFileName2[20]  = UNC_DISK2; /* UNC Name of disk 2 */
   PBYTE  pBuffer        = 0;
   ULONG  cbTotal        = 0;

   APIRET rc             = NO_ERROR;            /* Return code */

   /* Open a raw file system disk #1 for reading */
   rc = DosOpen(uchFileName1,               /* File name */
                hfDisk1,                   /* File handle */
                ulAction,                  /* Action taken by DosOpen */
                0L,                         /* no file size */
                FILE_NORMAL,                /* File attribute */
                OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
                OPEN_SHARE_DENYNONE |       /* Access mode */
                OPEN_ACCESS_READONLY,
                0L);                        /* No extented attributes */
   if (rc != NO_ERROR) {
      printf("DosOpen error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Set the pointer to the begining of the disk */
   rc = DosSetFilePtr(hfDisk1,      /* Handle for disk 1 */
                      0L,           /* Offset must be multiple of 512 */
                      FILE_BEGIN,   /* Begin of the disk */
                      ulLocation); /* New pointer location */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Open a raw file system disk #2 for writing */
   rc = DosOpen(uchFileName2,               /* File name */
                hfDisk2,                   /* File handle */
                ulAction,                  /* Action taken by DosOpen */
                0L,                         /* no file size */
                FILE_NORMAL,                /* File attribute */
                OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
                OPEN_SHARE_DENYNONE |       /* Access mode */
                OPEN_ACCESS_READWRITE,
                0L);                        /* No extented attributes */
   if (rc != NO_ERROR) {
      printf("DosOpen error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Set the pointer to the begining of the disk */
   rc = DosSetFilePtr(hfDisk2,      /* Handle for disk 1 */
                      0L,           /* Offset must be multiple of 512 */
                      FILE_BEGIN,   /* Begin of the disk */
                      ulLocation); /* New pointer location */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error rc = %u\n", rc);
      return(1);
   } /* endif */


   /* Allocate 64K of memory for transfer operations */
   rc = DosAllocMem((PPVOID)pBuffer, /* Pointer to buffer */
                     SIXTY_FOUR_K,      /* Buffer size */
                     PAG_COMMIT |     /* Allocation flags */
                     PAG_READ |
                     PAG_WRITE);
   if (rc != NO_ERROR) {
      printf("DosAllocMem error rc = %u\n", rc);
      return(1);
   } /* endif */

   cbRead = SIXTY_FOUR_K;
   while (rc == NO_ERROR  cbTotal  TEN_MEG) {
      /* Read from #1 */
      rc = DosRead(hfDisk1,         /* Handle for disk 1 */
                   pBuffer,         /* Pointer to buffer */
                   cbRead,          /* Size must be multiple of 512 */
                   cbActualRead);  /* Actual read by DosOpen */
      if (rc) {
         printf("DosRead error return code = %u\n", rc);
         return 1;
      }

      /* Write to disk #2 */
      cbWrite = cbActualRead;
      rc = DosWrite(hfDisk2,         /* Handle for disk 2 */
                    pBuffer,         /* Pointer to buffer */
                    cbWrite,         /* Size must be multiple of 512 */
                    cbActualWrote); /* Actual written by DosOpen */
      if (rc) {
         printf("DosWrite error return code = %u\n", rc);
         return 1;
      }
      if (cbActualRead != cbActualWrote) {
         printf("Bytes read (%u) does not equal bytes written (%u)\n",
                cbActualRead, cbActualWrote);
         return 1;
      }
      cbTotal += cbActualRead; /* Update total transferred */
   }

   printf("Transfer successfully %d bytes from disk #1 to disk #2.\n",
          cbTotal);

   /* Free allocated memmory */
   rc = DosFreeMem(pBuffer);
   if (rc != NO_ERROR) {
      printf("DosFreeMem error return code = %u\n", rc);
      return 1;
   }

   rc = DosClose(hfDisk1);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
   }

   rc = DosClose(hfDisk2);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
return NO_ERROR;
}
```

DosOpenL

Purpose

DosOpenL opens a new file, an existing file, or a replacement for an existing file. An open file can have extended attributes.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosOpenL **(PSZ pszFileName, PHFILE pHf, PULONG pulAction, LONGLONG cbFile, ULONG ulAttribute, ULONG fsOpenFlags, ULONG fsOpenMode, PEAOP2 peaop2)**

Parameters

pszFileName PSZ) input
Address of the ASCIIZ path name of the file or device to be opened.

pHf PHFILE) output

Address of the handle for the file.

pulAction PULONG) output

Address of the variable that receives the value that specifies the action taken by the DosOpenL function.

If DosOpenL fails, this value has no meaning. Otherwise, it is one of the following values

1	FILE_EXISTED File already existed.
2	FILE_CREATED File was created.
3	FILE_TRUNCATED File existed and was changed to a given size (file was replaced).

cbFile LONGLONG) input

New logical size of the file (end of data, EOD), in bytes.

This parameter is significant only when creating a new file or replacing an existing one. Otherwise, it is ignored. It is an error to create or replace a file with a nonzero length if the *fsOpenMode* Access-Mode flag is set to read-only.

ulAttribute ULONG) input

File attribute information.

Possible values are

Bits	Description
31 6	Reserved, must be 0.
5	FILE_ARCHIVED (0x00000020) File has been archived.
4	FILE_DIRECTORY (0x00000010) File is a subdirectory.
3	Reserved, must be 0.
2	FILE_SYSTEM (0x00000004) File is a system file.
1	FILE_HIDDEN (0x00000002) File is hidden and does not appear in a directory listing.
0	FILE_READONLY (0x00000001) File can be read from, but not written to.
0	FILE_NORMAL (0x00000000) File can be read from or written to.

File attributes apply only if the file is created.

These bits may be set individually or in combination. For example, an attribute value of 0x00000021 (bits 5 and 0 set to 1) indicates a read-only file that has been archived.

fsOpenFlags ULONG) input

The action to be taken depending on whether the file exists or does not exist.

Possible values are

Bits	Description
31 8	Reserved, must be 0.

7 4	The following flags apply if the file does not exist	
	0000	OPEN_ACTION_FAIL_IF_NEW Open an existing file; fail if the file does not exist.
	0001	OPEN_ACTION_CREATE_IF_NEW Create the file if the file does not exist.
3 0	The following flags apply if the file already exists	
	0000	OPEN_ACTION_FAIL_IF_EXISTS Open the file; fail if the file already exists.
	0001	OPEN_ACTION_OPEN_IF_EXISTS Open the file if it already exists.
	0010	OPEN_ACTION_REPLACE_IF_EXISTS Replace the file if it already exists.

fsOpenMode ULONG)   input
The mode of the open function. Possible values are

Bits	Description
31 16	Reserved, must be zero.
15	OPEN_FLAGS_DASD (0x00008000) Direct Open flag 0 — *pszFileName* represents a file to be opened normally. 1 — *pszFileName* is drive (such as c or a), and represents a mounted disk or diskette volume to be opened for direct access.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000) Write-Through flag 0 — Writes to the file may go through the file-system driver s cache. The file-system driver writes the sectors when the cache is full or the file is closed. 1 — Writes to the file may go through the file-system driver s cache, but the sectors are written (the actual file I/O operation is completed) before a synchronous write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit must be set, because the data must be written to the medium for synchronous write operations. This bit flag is not inherited by child processes.
13	OPEN_FLAGS_FAIL_ON_ERROR (0x00002000) Fail-Errors flag. Media I/O errors are handled as follows 0 — Reported through the system critical-error handler. 1 — Reported directly to the caller by way of a return code. Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands always get reported directly to the caller by way of return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O calls. This flag bit is not inherited by child processes.
12	OPEN_FLAGS_NO_CACHE (0x00001000) No-Cache Cache flag

	0	The file-system driver should place data from I/O operations into its cache.
	1	I/O operations to the file need not be done through the file-system driver s cache.
	The setting of this bit determines whether file-system drivers should place data into the cache. Like the write-through bit, this is a per-handle bit, and is not inherited by child processes.	
11	Reserved; must be 0.	
10 8	The locality of reference flags contain information about how the application is to get access to the file. The values are as follows	
	000	OPEN_FLAGS_NO_LOCALITY (0x00000000) No locality known.
	001	OPEN_FLAGS_SEQUENTIAL (0x00000100) Mainly sequential access.
	010	OPEN_FLAGS_RANDOM (0x00000200) Mainly random access.
	011	OPEN_FLAGS_RANDOMSEQUENTIAL (0x00000300) Random with some locality.
7	OPEN_FLAGS_NOINHERIT (0x00000080) Inheritance flag	
	0	File handle is inherited by a process created from a call to DosExecPgm.
	1	File handle is private to the current process.
	This bit is not inherited by child processes.	
6 4	Sharing Mode flags. This field defines any restrictions to file access placed by the caller on other processes. The values are as follows	
	001	OPEN_SHARE_DENYREADWRITE (0x00000010) Deny read write access.
	010	OPEN_SHARE_DENYWRITE (0x00000020) Deny write access.
	011	OPEN_SHARE_DENYREAD (0x00000030) Deny read access.
	100	OPEN_SHARE_DENYNONE (0x00000040) Deny neither read nor write access (deny none).
	Any other value is invalid.	
29	OPEN_SHARE_DENYLEGACY (0x10000000) Deny read/write access by the DosOpen command.	
	0	Allow read/write access by the DosOpen command.
	1	Deny read/write access by the DosOpen command. A file opened by DosOpenL will not be allowed to grow larger than 2GB while that same file is open with a legacy DosOpen call. Setting this bit to 1 will prevent access by the obsolete DosOpen API and ensure that no error will occur when

growing the file.

Any other value is invalid.

3	Reserved; must be 0.
2 0	Access-Mode flags. This field defines the file access required by the caller. The values are as follows

000	OPEN_ACCESS_READONLY (0x00000000) Read-only access
001	OPEN_ACCESS_WRITEONLY (0x00000001) Write-only access
010	OPEN_ACCESS_READWRITE (0x00000002) Read/write access.

Any other value is invalid, as are any other combinations.

File sharing requires the cooperation of sharing processes. This cooperation is communicated through sharing and access modes. Any sharing restrictions placed on a file opened by a process are removed when the process closes the file with a DosClose request.

Sharing Mode
: Specifies the type of file access that other processes may have. For example, if other processes can continue to read the file while your process is operating on it, specify Deny Write. The sharing mode prevents other processes from writing to the file but still allows them to read it.

Access Mode
: Specifies the type of file access (access mode) needed by your process. For example, if your process requires read/write access, and another process has already opened the file with a sharing mode of Deny None, your DosOpenL request succeeds. However, if the file is open with a sharing mode of Deny Write, the process is denied access.

If the file is inherited by a child process, all sharing and access restrictions also are inherited.

If an open file handle is duplicated by a call to DosDupHandle, all sharing and access restrictions also are duplicated.

peaop2 PEAOP2) in/out

Extended attributes.

This parameter is used only to specify extended attributes (EAs) when creating a new file, replacing an existing file, or truncating an existing file. When opening existing files, it should be set to null.

Input	The address of the extended-attribute buffer, which contains an EAOP2 structure. *fpFEA2List* points to a data area where the relevant FEA2 list is to be found. *fpGEA2List* and *oError* are ignored.
Output	*fpGEA2List* and *fpFEA2List* are unchanged. The area that *fpFEA2List* points to is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 entry where the error occurred. The return code from DosOpenL is the error code for that error condition. If no error occurred, *oError* is undefined.

If *peaop2* is zero, then no extended attributes are defined for the file.

If extended attributes are not to be defined or modified, the pointer *peaop2* must be set to zero.

Returns

ulrc APIRET) returns

Return Code.

DosOpenL returns one of the following values

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND

3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES
5	ERROR_ACCESS_DENIED
12	ERROR_INVALID_ACCESS
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
82	ERROR_CANNOT_MAKE
87	ERROR_INVALID_PARAMETER
99	ERROR_DEVICE_IN_USE
108	ERROR_DRIVE_LOCKED
110	ERROR_OPEN_FAILED
112	ERROR_DISK_FULL
206	ERROR_FILENAME_EXCED_RANGE
231	ERROR_PIPE_BUSY

Remarks

A successful DosOpenL request returns a handle for accessing the file. The read/write pointer is set at the first byte of the file. The position of the pointer can be changed with DosSetFilePtrL or by read and write operations on the file.

The file s date and time can be queried with DosQueryFileInfo. They are set with DosSetFileInfo.

The read-only attribute of a file can be set with the ATTRIB command.

ulAttribute cannot be set to Volume Label. To set volume label information, issue DosSetFSInfo with a logical drive number. Volume labels cannot be opened.

cbFile affects the size of the file only when the file is new or is a replacement. If an existing file is opened, *cbFile* is ignored. To change the size of the existing file, issue DosSetFileSizeL.

The value in *cbFile* is a recommended size. If the full size cannot be allocated, the open request may still succeed. The file system makes a reasonable attempt to allocate the new size in an area that is as nearly contiguous as possible on the medium. When the file size is extended, the values of the new bytes are undefined.

The Direct Open bit provides direct access to an entire disk or diskette volume, independent of the file system. This mode of opening the volume that is currently on the drive returns a handle to the calling function; the handle represents the logical volume as a single file. The calling function specifies this handle with a DosDevIOCtl Category 8, DSK_LOCKDRIVE request to prevent other processes from accessing the logical volume. When you are finished using the logical volume, issue a DosDevIOCtl Category 8, DSK_UNLOCKDRIVE request to allow other processes to access the logical volume.

The file-handle state bits can be set by DosOpenL and DosSetFHState. An application can query the file-handle state bits, as well as the rest of the Open Mode field, by issuing DosQueryFHState.

You can use an EAOP2 structure to set extended attributes in *peaop2* when creating a file, replacing an existing file, or truncating an existing file. No extended attributes are set when an existing file is just opened.

A replacement operation is logically equivalent to atomically deleting and re-creating the file. This means that any extended attributes associated with the file also are deleted before the file is re-created.

Related Functions

- DosClose
- DosDevIOCtl
- DosDupHandle
- DosQueryHType
- DosSetFileInfo

- DosSetFilePtrL
- DosSetFileSizeL
- DosSetMaxFH
- DosSetRelMaxFH

Example Code

This example opens or creates and opens a normal file named DOSTEST.DAT , writes to it, reads from it, and finally closes it.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include os2.h
#include stdio.h
#include string.h

int main(void)
HFILE  hfFileHandle   = 0L;     /* Handle for file being manipulated */
ULONG  ulAction       = 0;      /* Action taken by DosOpenL */
ULONG  ulBytesRead    = 0;      /* Number of bytes read by DosRead */
ULONG  ulWrote        = 0;      /* Number of bytes written by DosWrite */
LONGLONG  ullLocal    = 0;      /* File pointer position after DosSetFilePtrL */
UCHAR  uchFileName20  = "dostest.dat",     /* Name of file */
uchFileData100 = " ";              /* Data to write to file */
APIRET rc             = NO_ERROR;            /* Return code */

/* Open the file test.dat.  Use an existing file or create a new */
/* one if it doesn't exist.                                      */
rc = DosOpenL(uchFileName,                    /* File path name */
hfFileHandle,                  /* File handle */
ulAction,                      /* Action taken */
(LONGLONG)100,                    /* File primary allocation */
FILE_ARCHIVED | FILE_NORMAL,    /* File attribute */
OPEN_ACTION_CREATE_IF_NEW |
OPEN_ACTION_OPEN_IF_EXISTS,     /* Open function type */
OPEN_FLAGS_NOINHERIT |
OPEN_SHARE_DENYNONE  |
OPEN_ACCESS_READWRITE,          /* Open mode of the file */
0L);                            /* No extended attribute */

if (rc != NO_ERROR)
printf("DosOpenL error return code = %u\n", rc);
return 1;
 else
printf ("DosOpenL Action taken = %ld\n", ulAction);
 /* endif *//* Write a string to the file */
strcpy (uchFileData, "testing...\n1...\n2...\n3\n");

rc = DosWrite (hfFileHandle,                /* File handle */
(PVOID) uchFileData,         /* String to be written */
sizeof (uchFileData),        /* Size of string to be written */
ulWrote);                   /* Bytes actually written */

if (rc != NO_ERROR)
printf("DosWrite error return code = %u\n", rc);
return 1;
 else
printf ("DosWrite Bytes written = %u\n", ulWrote);
 /* endif */

/* Move the file pointer back to the beginning of the file */
rc = DosSetFilePtrL (hfFileHandle,           /* File Handle */
(LONGLONG)0,            /* Offset */
FILE_BEGIN,             /* Move from BOF */
ullLocal);             /* New location address */
if (rc != NO_ERROR)
printf("DosSetFilePtrL error return code = %u\n", rc);
return 1;


/* Read the first 100 bytes of the file */
rc = DosRead (hfFileHandle,               /* File Handle */
uchFileData,                 /* String to be read */
100L,                        /* Length of string to be read */
ulBytesRead);               /* Bytes actually read */

if (rc != NO_ERROR)
printf("DosRead error return code = %u\n", rc);
return 1;
 else
```

```
printf ("DosRead Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
 /* endif */

rc = DosClose(hfFileHandle);                /* Close the file */

if (rc != NO_ERROR)
printf("DosClose error return code = %u\n", rc);
return 1;

return NO_ERROR;
```

DosPerfSysCall

Purpose

DosPerfSysCall retrieves system performance information and performs software tracing.

Syntax

```
#define INCL_BASE
#include os2.h>
```

APIRET DosPerfSysCall **(ULONG ulCommand, ULONG ulParm1, ULONG ulParm1,ULONG ulParm2, ULONG ulParm3)**

Parameters

ulCommand (ULONG) input
Accepts following commands

CMD_KI_RDCNT 0x63	Reads CPU utilization information in both uniprocessor and symmetric multi-processor (SMP) environments by taking a snapshot of the time stamp counters. To determine CPU utilization, the application must compute the difference between two time stamp snapshots using 64 bit aritimetic. See the example code for details.
CMD_SOFTTRACE_LOG0x14	Records software trace information.

ulParm1 (ULONG) input/output

CMD_KI_RDCNT	Pointer to CPUUTIL structure *ulParm1* would be set to the address of the CPUUTIL structure. *ulParm2* and *ulParm3* are not used and should be set to zero.
CMD_SOFTTRACE_LOG	Major code for the trace entry in the range of 0 to 255. Major codes 184 (0x00b8) and 185 (0x00b9) have been reserved for customer use. Major code 1 is reserved for exclusive use by IBM(R).

ulParm2 (ULONG) input/output

CMD_KI_RDCNT	0 (reserved)
CMD_SOFTTRACE_LOG	Minor code for the trace entry in the range of 0 to 255.

ulParm3 (ULONG) input/output

CMD_KI_RDCNT	0 (reserved)
CMD_SOFTTRACE_LOG	Pointer to a HOOKDATA data structure.

Returns

ulrc (APIRET) returns
Return Code.

DosPerfSysCall returns one of the following values

0	NO_ERROR
1	ERROR_INVALID_FUNCTION

Remarks

DosPerfSysCall is a general purpose performance function. This function accepts four parameters. The first parameter is the command requested. The other three parameters are command specific.

Some functions of DosPerfSysCall may have a dependency on Intel Pentium or Pentium-Pro support. If a function cannot be provided because OS/2 is not running on a processor with the required features, a return code will indicate an attempt to use an unsupported function.

Example Code

This example uses DosPerfSysCall to obtain CPU utilization information.

```
#define INCL_BASE
#include os2.h>
#include stdio.h>
#include stdlib.h>
#include string.h>
#include perfutil.h>

#define LL2F (high, low) (4294967296.0* (high) + (low)

void main (int argc, char *argv[])
{

   APIRET   rc;
   int      i, iter, sleep_sec;
   double   ts_val, idle_val_prev;
   double   idle_val, busy_val_prev;
   double   busy_val, busy_val_prev;
   dobule   intr_val intr_val_prev;
   CPUUTIL  CPUUtil;

   if ((argc  2) || (*aargv[1]  '1') || (*aargv[1] > '9')) {
     fprintf(stderr, "usage %s [1-9]\n", argv[0]);
     exit(0);
   }
   sleep_sec = *argv[1] - '0';

   iter = 0;
   do {
     rec = DosPerfSysCall (CMD_KI_RDCNT, (ULONG) CPUUtil,0,0);
     if (rc) {
        fprintf (stderr, "CMD_KI_RDCNT failed rc = %d\n", rc);
        exit(1);
     }
     ts_val = LL2F (CPUUtil.ulTimeHigh, CPUUtil.ulTimeLow);
     idle_val = LL2f (CPUUtil.ulIdleHigh, CPUUtil.ulIdleLow);
     busy_val = LL2F (CPUUtil.ulBusyHigh, CPUUtil.ulBusyLow);
     intr_val = LL2F (CPUUtil.ulIntrHigh, CPUUtil.ulIntrLow);

     if (iter > 0) {
        double ts_delta = ts_val - ts_val_prev;
        printf ("idle %4.2%% busy %4.2f%% intr %4.2f%%\n";
           (idle_val - idle_val_prev/ts_delta*100.0,
           (busy_val - busy_val_prev/ts_delta*100.0,
           (intr_val - intr_val_prev/ts_delta*100.0);
     }
     ts_val_prev = ts_val;
     idle_val_prev = idle_val;
     busy_val_prev = busy_val;
     intr_val_prev = intr_val;

     iter++;
     DosSleep(1000*sleep_sec);

   } while (1);
}
```

This example performs software tracing from a program in ring 3.

```
#define INCL_BASE
#include os2.h>
#include stdio.h>
#include stdlib.h>
#include string.h>
#include perfutil.h>

int main (int argc, char *argv[])
{
   APIRET     rc;
   BYTE       HookBuffer [256];
   HOOKDATA   Hookdata = {0,HookBuffer};
   ULONG      ulMajor, ulMinor;
   *((PULONG) HookBuffer[0]) = 1;
   *((PULONG) HookBuffer[4]) = 2;
   *((PULONG) HookBuffer[8]) = 3;
   strcpy((PSZ HookBuffer[12], "Test of 3 ULONG values and a string.")
   HookData.ulLength = 12 + strlen((PSZHookBuffer[12]) + 1;

   ulMajor = 0x00b8
   ulMinor = 0x0001

   rc = DosPerfSystCall(CMD_SOFTTRACE_LOG, ulMajor, ulMinor, (ULONG) HookData);
   if (rc != NO_ERROR) {
     fprintf (stderr, "CMD_SOFTTRACE_LOG failed  rc = %u\n", rc);
     return 1;
     }

   return NO_ERROR;
}
```

DosProtectOpenL

Purpose

DosProtectOpenL opens a new file, an existing file, or a replacement for an existing file and returns a protected file handle. An open file can have extended attributes.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosProtectOpenL **(PSZ pszFileName, PHFILE phf, PULONG pulAction, LONGLONG cbFile, ULONG ulAttribute, ULONG fsOpenFlags, ULONG fsOpenMode, PEAOP2 peaop2, PFHLOCK pfhFileHandleLockID)**

Parameters

pszFileName PSZ)   input

Address of the ASCIIZ path name of the file or device to be opened.

phf PHFILE)   output

Address of the handle for the file.

pulAction PULONG)   output

A pointer to the ULONG in which the value that specifies the action taken by DosProtectOpenL is returned.

If DosProtectOpenL fails, this value has no meaning. Otherwise, it is one of the following values

1 FILE_EXISTED

File already existed.

2 FILE_CREATED

File was created.

3 FILE_TRUNCATED

File existed and was changed to a given size (file was replaced).

cbFile LONGLONG)   input

New logical size of the file (end of data, EOD), in bytes.

This parameter is significant only when creating a new file or replacing an existing one. Otherwise, it is ignored. It is an error to create or replace a file with a nonzero length if the *fsOpenMode* Access-Mode flag is set to read-only.

ulAttribute ULONG)   input

File attributes.

This parameter contains the following bit fields

Bits	Description
31 6	Reserved, must be 0.
5	FILE_ARCHIVED (0x00000020) File has been archived.
4	FILE_DIRECTORY (0x00000010) File is a subdirectory.
3	Reserved, must be 0.
2	FILE_SYSTEM (0x00000004) File is a system file.
1	FILE_HIDDEN (0x00000002) File is hidden and does not appear in a directory listing.
0	FILE_READONLY (0x00000001) File can be read from, but not written to.
0	FILE_NORMAL (0x00000000) File can be read from or written to.

File attributes apply only if the file is created.

These bits may be set individually or in combination. For example, an attribute value of 0x00000021 (bits 5 and 0 set to 1) indicates a read-only file that has been archived.

fsOpenFlags ULONG)   input

The action to be taken depending on whether the file exists or does not exist.

This parameter contains the following bit fields

Bits	Description	
31 8	Reserved, must be 0.	
7 4	The following flags apply if the file does not exist	
	0000	OPEN_ACTION_FAIL_IF_NEW Open an existing file; fail if the file does not exist.
	0001	OPEN_ACTION_CREATE_IF_NEW Create the file if the file does not exist.
3 0	The following flags apply if the file does not exist	
	0000	OPEN_ACTION_FAIL_IF_EXISTS Open the file; fail if the file already exists.
	0001	OPEN_ACTION_OPEN_IF_EXISTS

Open the file if it already exists.

0010	OPEN_ACTION_REPLACE_IF_EXISTS
	Replace the file if it already exists.

fsOpenMode ULONG)   input
The mode of the open function.

This parameter contains the following bit fields

Bits	Description
29 16	Reserved, must be zero.
30	OPEN_FLAGS_PROTECTED_HANDLE (0x40000000) Protected file handle flag. 0 Unprotected Handle 1 Protected Handle Protected handle requires the *pfhFileHandleLockID* to be specified on subsequent DosProtect*xxxx* calls. Unprotected handle requires the *pfhFileHandleLockID* value to be specified as zero on subsequent DosProtect*xxxx* calls. An unprotected handle may be used with the unprotected calls such as DosRead and DosWrite.
31	Reserved, must be zero.
15	OPEN_FLAGS_DASD (x00008000) Direct Open flag 0 *pszFileName* represents a file to be opened normally. 1 *pszFileName* is drive (such as C or A), and represents a mounted disk or diskette volume to be opened for direct access.
14	OPEN_FLAGS_WRITE_THROUGH (0x00004000) Write-Through flag 0 Writes to the file may go through the file-system driver s cache. The file-system driver writes the sectors when the cache is full or the file is closed. 1 Writes to the file may go through the file-system driver s cache, but the sectors are written the actual file I O operation is completed) before a synchronous write call returns. This state of the file defines it as a synchronous file. For synchronous files, this bit must be set, because the data must be written to the medium for synchronous write operations. This bit flag is not inherited by child processes.
13	OPEN_FLAGS_FAIL_ON_ERROR (0x00002000) Fail-Errors flag. Media I O errors are handled as follows 0 Reported through the system critical-error handler. 1 Reported directly to the caller by way of a return code. Media I/O errors generated through Category 08h Logical Disk Control IOCtl Commands always get reported directly to the caller by way of return code. The Fail-Errors function applies only to non-IOCtl handle-based file I/O calls. This flag bit is not inherited by child processes.
12	OPEN_FLAGS_NO_CACHE (0x00001000) No-Cache/Cache flag

	0	The file-system driver should place data from I O operations into its cache.	
	1	I/O operations to the file need not be done through the file-system driver s cache.	

The setting of this bit determines whether file-system drivers should place data into the cache. Like the write-through bit, this is a per-handle bit, and is not inherited by child processes.

11 Reserved; must be 0.

10 8 The locality of reference flags contain information about how the application is to get access to the file. The values are as follows

000	OPEN_FLAGS_NO_LOCALITY (0x00000000) No locality known.
001	OPEN_FLAGS_SEQUENTIAL (0x00000100) Mainly sequential access.
010	OPEN_FLAGS_RANDOM (0x00000200) Mainly random access.
011	OPEN_FLAGS_RANDOMSEQUENTIAL (0x00000300) Random with some locality.

7 OPEN_FLAGS_NOINHERIT (0x00000080)

Inheritance flag

0	File handle is inherited by a process created from a call to DosExecPgm.
1	File handle is private to the current process.

This bit is not inherited by child processes.

6 4 Sharing Mode flags. This field defines any restrictions to file access placed by the caller on other processes. The values are as follows

001	OPEN_SHARE_DENYREADWRITE (0x00000010) Deny read write access.
010	OPEN_SHARE_DENYWRITE (0x00000020) Deny write access.
011	OPEN_SHARE_DENYREAD (0x00000030) Deny read access.
100	OPEN_SHARE_DENYNONE (0x00000040) Deny neither read nor write access (deny none).

29 OPEN_SHARE_DENYLEGACY (0x10000000)

Deny read/write access by the DosOpen command

0	Allow read/write access by the DosOpen command.
1	Deny read/write access by the DosOpen command. A file opened by DosOpenL will not be allowed to grow larger than 2GB while that same file is open via a legacy DosOpen call. Setting this bit to 1 will prevent access by the obsolete DosOpen API and ensure that no error will occur when growing the file.

	Any other value is invalid.	
3	Reserved; must be 0.	
2 0	Access-Mode flags. This field defines the file access required by the caller. The values are as follows	
	000	OPEN_ACCESS_READONLY (0x00000000) Read-only access
	001	OPEN_ACCESS_WRITEONLY (0x00000001) Write-only access
	010	OPEN_ACCESS_READWRITE (0x00000002) Read/write access.
	Any other value is invalid, as are any other combinations.	

File sharing requires the cooperation of sharing processes. This cooperation is communicated through sharing and access modes. Any sharing restrictions placed on a file opened by a process are removed when the process closes the file with a DosClose request.

Sharing Mode

Specifies the type of file access that other processes may have. For example, if other processes can continue to read the file while your process is operating on it, specify Deny Write. The sharing mode prevents other processes from writing to the file but still allows them to read it.

Access Mode

Specifies the type of file access (access mode) needed by your process. For example, if your process requires read/write access, and another process has already opened the file with a sharing mode of Deny None, your DosProtectOpenL request succeeds. However, if the file is open with a sharing mode of Deny Write, the process is denied access.

If the file is inherited by a child process, all sharing and access restrictions also are inherited.

If an open file handle is duplicated by a call to DosDupHandle, all sharing and access restrictions also are duplicated.

peaop2 PEAOP2)   in/out

A pointer to an extended attribute buffer.

Input	The address of the extended-attribute buffer, which contains an EAOP2 structure. The *fpFEA2List* field in the EAOP2 structure points to a data area where the relevant FEA2 list is to be found. The *fpGEA2List* and *oError* fields are ignored.
Output	*fpGEA2List* and *fpFEA2List* are unchanged. The area that *fpFEA2List* points to is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 entry where the error occurred. The return code from DosProtectOpenL is the error code for that error condition. If no error occurred, *oError* is undefined.

If *peaop2* is zero, then no extended attributes are defined for the file. If extended attributes are not to be defined or modified, the pointer *peaop2* must be set to zero.

pfhFileHandleLockID PFHLOCK)   output

The address of the 32-bit lockid for the file handle.

Returns

ulrc APIRET)   returns

Return Code.

DosProtectOpenL returns one of the following values

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
4	ERROR_TOO_MANY_OPEN_FILES

5	ERROR_ACCESS_DENIED
12	ERROR_INVALID_ACCESS
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
82	ERROR_CANNOT_MAKE
87	ERROR_INVALID_PARAMETER
99	ERROR_DEVICE_IN_USE
108	ERROR_DRIVE_LOCKED
110	ERROR_OPEN_FAILED
112	ERROR_DISK_FULL
206	ERROR_FILENAME_EXCED_RANGE
231	ERROR_PIPE_BUSY

Remarks

A successful DosProtectOpenL request returns a handle and a 32-bit lockid for accessing the file. The read/write pointer is set at the first byte of the file. The position of the pointer can be changed with DosProtectSetFilePtrL or by read and write operations on the file.

The file s date and time can be queried with DosProtectQueryFileInfo. They are set with DosProtectSetFileInfo.

The read-only attribute of a file can be set with the ATTRIB command.

ulAttribute cannot be set to Volume Label. To set volume-label information, issue DosProtectSetFileInfo with a logical drive number. Volume labels cannot be opened.

cbFile affects the size of the file only when the file is new or is a replacement. If an existing file is opened, *cbFile* is ignored. To change the size of the existing file, issue DosProtectSetFileSizeL.

The value in *cbFile* is a recommended size. If the full size cannot be allocated, the open request may still succeed. The file system makes a reasonable attempt to allocate the new size in an area that is as nearly contiguous as possible on the medium. When the file size is extended, the values of the new bytes are undefined.

The Direct Open bit provides direct access to an entire disk or diskette volume, independent of the file system. This mode of opening the volume that is currently on the drive returns a handle to the calling function; the handle represents the logical volume as a single file. The calling function specifies this handle with a DosDevIOCtl Category 8, DSK_LOCKDRIVE request to prevent other processes from accessing the logical volume. When you are finished using the logical volume, issue a DosDevIOCtl Category 8, DSK_UNLOCKDRIVE request to allow other processes to access the logical volume.

The file-handle state bits can be set by DosProtectOpenL and DosProtectSetFHState. An application can query the file-handle state bits, as well as the rest of the Open Mode field, by issuing DosProtectQueryFHState.

You can use an EAOP2 structure to set extended attributes in *peaop2* when creating a file, replacing an existing file, or truncating an existing file. No extended attributes are set when an existing file is just opened.

A replacement operation is logically equivalent to atomically deleting and re-creating the file. This means that any extended attributes associated with the file also are deleted before the file is re-created.

The *pfhFileHandleLockID* returned is required on each of the DosProtectxxx functions. An incorrect *pfhFileHandleLockID* on subsequent DosProtectxxx calls results in an ERROR_ACCESS_DENIED return code.

The DosProtectxxx functions can be used with a NULL filehandle lockid, if the subject filehandle was obtained from DosOpen.

Related Functions

- DosDevIOCtl
- DosDupHandle
- DosProtectClose
- DosProtectSetFileInfo
- DosProtectSetFilePtrL

- DosProtectSetFileSizeL
- DosQueryHType
- DosSetMaxFH
- DosSetRelMaxFH

Example Code

This example opens or creates and opens a file named DOSPROT.DAT , writes to it, reads from it, and finally closes it using DosProtect functions.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include os2.h
#include stdio.h
#include string.h

int main(VOID)
HFILE  hfFileHandle   = 0L;
ULONG  ulAction       = 0;
ULONG  ulBytesRead    = 0;
ULONG  ulWrote        = 0;
LONGLONG  ullLocal    = 0;
UCHAR  uchFileName20  = "dosprot.dat",
uchFileData100 = " ";
FHLOCK FileHandleLock = 0;        /* File handle lock   */
APIRET rc             = NO_ERROR; /* Return code */

/* Open the file dosprot.dat.  Make it read/write, open it */
/* if it already exists and create it if it is new.     */
rc = DosProtectOpenL(uchFileName,             /* File path name          */
hfFileHandle,                  /* File handle             */
ulAction,                      /* Action taken            */
(LONGLONG)100,                      /* File primary allocation */
FILE_ARCHIVED | FILE_NORMAL,    /* File attribute          */
OPEN_ACTION_CREATE_IF_NEW |
OPEN_ACTION_OPEN_IF_EXISTS,     /* Open function type      */
OPEN_FLAGS_NOINHERIT |
OPEN_SHARE_DENYNONE  |
OPEN_ACCESS_READWRITE,          /* Open mode of the file   */
0L,                             /* No extended attribute   */
FileHandleLock);               /* File handle lock id     */
if (rc != NO_ERROR)
printf("DosProtectOpenL error return code = %u\n", rc);
return 1;
 else
printf ("DosProtectOpenL Action taken = %u\n", ulAction);
 /* endif */

/* Write a string to the file */
strcpy (uchFileData, "testing...\n3...\n2...\n1\n");

rc = DosProtectWrite (hfFileHandle,       /* File handle                  */
(PVOID) uchFileData,       /* String to be written         */
sizeof (uchFileData),      /* Size of string to be written */
ulWrote,                 /* Bytes actually written       */
FileHandleLock);           /* File handle lock id          */if (rc != NO_ERROR)
printf("DosProtectWrite error return code = %u\n", rc);
return 1;
 else
printf ("DosProtectWrite Bytes written = %u\n", ulWrote);
 /* endif */

/* Move the file pointer back to the beginning of the file */
rc = DosProtectSetFilePtrL (hfFileHandle,    /* File Handle          */
(LONGLONG)0,            /* Offset               */
FILE_BEGIN,             /* Move from BOF        */
ullLocal,               /* New location address */
FileHandleLock);        /* File handle lock id  */
if (rc != NO_ERROR)
printf("DosSetFilePtrL error return code = %u\n", rc);
return 1;


/* Read the first 100 bytes of the file */
rc = DosProtectRead (hfFileHandle,         /* File Handle                 */
uchFileData,                 /* String to be read           */
100L,                        /* Length of string to be read */
ulBytesRead,                /* Bytes actually read          */
FileHandleLock);             /* File handle lock id          */
if (rc != NO_ERROR)
printf("DosProtectRead error return code = %u\n", rc);
return 1;
 else
printf("DosProtectRead Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
 /* endif */

rc = DosProtectClose(hfFileHandle, FileHandleLock);   /* Close the file */
if (rc != NO_ERROR)
printf("DosProtectClose error return code = %u\n", rc);
return 1;

return NO_ERROR;
```

DosProtectQueryFileInfo

Purpose

DosProtectQueryFileInfo gets file information.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosProtectQueryFileInfo **(HFILE hf, ULONG ulInfoLevel, PVOID pInfo, ULONG cbInfoBuf, FHLOCK fhFileHandleLockID)**

Parameters

hf HFILE)   input

File handle.

ulInfoLevel ULONG)   input

Level of file information required.

Specify a value

1	FIL_STANDARD
	Level 1 file information
11	FIL_STANDARDL
	Level 11 file information
2	FIL_QUERYEASIZE
	Level 2 file information
12	FIL_QUERYEASIZEL
	Level 12 file information
3	FIL_QUERYEASFROMLIST
	Level 3 file information

The structures described in *pInfo* indicate the information returned for each of these levels.

pInfo PVOID)   output

Address of the storage area where the system returns the requested level of file information.

File information, where applicable, is at least as accurate as the most recent DosProtectClose, DosResetBuffer, DosProtectSetFileInfo, or DosSetPathInfo.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)

pInfo contains the FILESTATUS3 data structure, to which file information is returned.

Level 11 File Information (*ulInfoLevel* == FIL_STANDARDL)

pInfo contains the FILESTATUS3L data structure, to which file information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)

pInfo contains the FILESTATUS4 data structure. This is similar to the Level 1 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an ULONG. On output, this field contains the size, in bytes, of the file s entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel*. The buffer size is less than or equal to twice the size of the file s entire EA set on disk.

Level 12 File Information (*ulInfoLevel* == FIL_QUERYEASIZEL)

pInfo contains the FILESTATUS4L data structure. This is similar to the Level 11 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an ULONG. On output, this field contains the size, in bytes, of the file s entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel*. The buffer size is less than or equal to twice the size of the file s entire EA set on disk.

Level 3 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLIST)

Input	*pInfo* contains an EAOP2 data structure. *fpGEA2List* points to a GEA2 list defining the attribute names whose values are returned. The GEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the GEA2 list. The *oNextEntryOffset* field in the last entry of the GEA2 list must be zero. *fpFEA2List* points to a data area where the relevant FEA2 list is returned. The length field of this FEA2 list is valid, giving the size of the FEA2 list buffer. *oError* is ignored.
Output	*pInfo* is unchanged. The buffer pointed to by *fpFEA2List* is filled in with the returned information. If the buffer that *fpFEA2List* points to is not large enough to hold the returned information (the return code is ERROR_BUFFER_OVERFLOW), *cbList* is still valid, assuming there is at least enough space for it. Its value is the size of the entire EA set on disk for the file, even though only a subset of attributes was requested.

cbInfoBuf ULONG) input

The length, in bytes, of *pInfo*.

fhFileHandleLockID FHLOCK) input

The filehandle lockid returned by a previous DosProtectOpenL.

Returns

ulrc (APIRET) returns

Return Code.

DosProtectQueryFile returns one of the following values

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

Remarks

In the FAT file system, only date and time information contained in level-1 file information can be modified. Zero is returned for the creation and access dates and times.

To return information contained in any of the file information levels, DosProtectQueryFileInfo must be able to read the open file. DosProtectQueryFileInfo works only when the file is opened for read access, with a deny-write sharing mode specified for access by other processes. If another process that has specified conflicting sharing and access modes has already opened the file, any call to DosProtectOpen will fail.

DosProtectEnumAttribute returns the lengths of extended attributes. This information can be used to calculate what size *pInfo* needs to be to hold full-extended-attribute (FEA) information returned by DosProtectQueryFileInfo when Level 3 is specified. The size of the buffer is calculated as follows

Four bytes (for *oNextEntryOffset*) +
One byte (for *fEA*) +
One byte (for *cbName*) +
Two bytes (for *cbValue*) +
Value of *cbName* (for the name of the EA) +
One byte (for terminating NULL in *cbName*) +
Value of *cbValue* (for the value of the EA)

Related Functions

- DosProtectClose
- DosProtectOpenL
- DosProtectEnumAttribute
- DosProtectSetFileInfo
- DosQueryPathInfo
- DosResetBuffer
- DosProtectSetFileSizeL
- DosSetPathInfo

Example Code

This example creates a read-only file named DOSFDEL.DAT , then changes the file attributes to normal, and uses DosForceDelete to delete the file so that it can not be restored using UNDELETE.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include os2.h
#include stdio.h

int main(VOID)

UCHAR       uchFileName   = "DOSFDEL.DAT";   /* File to delete    */
HFILE       fhDelFile       = 0;               /* File handle from DosOpenL  */
FILESTATUS3L fsts3FileInfo   = 0;  /* Information associated with file   */
ULONG       ulBufferSize    = sizeof(FILESTATUS3L); /* File info buffer size */
ULONG       ulOpenAction    = 0;                 /* Action taken by DosOpenL */
APIRET      rc              = NO_ERROR;          /* Return code             */
FHLOCK      FHLock          = 0;                 /* File handle lock        */

/* Create a read-only file */

rc = DosProtectOpenL(uchFileName, fhDelFile,
ulOpenAction, (longlong)10, FILE_READONLY,
OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L, FHLock);
if (rc != NO_ERROR)
printf("DosProtectOpenL error return code = %u\n", rc);
return 1;


rc = DosProtectQueryFileInfo(fhDelFile, FIL_STANDARDL,
fsts3FileInfo, ulBufferSize, FHLock);   /* Get standard info */
if (rc != NO_ERROR)
printf("DosProtectQueryFileInfo error return code = %u\n", rc);
return 1;
 else  printf("File %s created read-only.\n",uchFileName);

/*   Change the file attributes to be "normal"  */
```

```
fsts3FileInfo.attrFile  = FILE_NORMAL;
rc = DosProtectSetFileInfo(fhDelFile, FIL_STANDARDL,
fsts3FileInfo, ulBufferSize, FHLock);
if (rc != NO_ERROR)
printf("DosProtectSetFileInfo error return code = %u\n", rc);
return 1;

rc = DosProtectClose(fhDelFile, FHLock);
/* Should verify that (rc != NO_ERROR) here... *//* Delete the file */

rc = DosForceDelete(uchFileName);
if (rc != NO_ERROR)
printf("DosForceDelete error return code = %u\n", rc);
return 1;
 else
printf("File %s has been deleted.\n",uchFileName);
  /* endif */

return NO_ERROR;
```

DosProtectSetFileInfo

Purpose

DosProtectSetFileInfo sets file information.

Syntax

```
#define INCLDOSFILEMGR
#include  os2.h
```

APIRET DosProtectSetFileInfo **(HFILE hf, ULONG ulInfoLevel, PVOID pInfoBuf, ULONG cbInfoBuf, FHLOCK fhFileHandleLockID)**

Parameters

hf HFILE)   input

File handle.

ulInfoLevel ULONG)   input

Level of file information being set.

Specify a value

1	FIL_STANDARD
	Level 1 file information
11	FIL_STANDARDL
	Level 11 file information
2	FIL_QUERYEASIZE
	Level 2 file information

The structures described in *pInfoBuf* indicate the information being set for each of these levels.

pInfoBuf PVOID)   input

Address of the storage area containing the structures for file information levels.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)

pInfoBuf contains the FILESTATUS3 data structure.

Level 11 File Information (*ulInfoLevel* == FIL_STANDARDL)

pInfo contains the FILESTATUS3L data structure, to which file information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfoBuf contains an EAOP2 data structure.

Level 2 sets a series of EA name/value pairs. On input, *pInfoBuf* is an EAOP2 data structure. *fpGEA2List* is ignored. *fpFEA2List* points to a data area where the relevant is an FEA2 list is to be found. *oError* is ignored.

On output, *fpGEA2List* and *fpFEA2List* are unchanged. The area pointed to by *fpFEA2List* is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 where the error occurred. The return code is the error code corresponding to the condition generating the error. If no error occurred, *oError* is undefined.

cbInfoBuf ULONG) input
The length, in bytes, of *pInfoBuf* .

fhFileHandleLockID FHLOCK) input
The filehandle lockid obtained from DosProtectOpen.

Returns

ulrc APIRET) returns
Return Code.

DosProtectSetFileInfo returns one of the following values

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
122	ERROR_INSUFFICIENT_BUFFER
124	ERROR_INVALID_LEVEL
130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

Remarks

DosProtectSetFileInfo is successful only when the file is opened for write access, and access by other processes is prevented by a deny-both sharing mode. If the file is already opened with conflicting sharing rights, any call to DosProtectOpen will fail.

A value of 0 in the date and time components of a field does not change the field. For example, if both last write date and last write time are specified as 0 in the Level 1 information structure, then both attributes of the file are left unchanged. If either last write date or last write time are other than 0, both attributes of the file are set to the new values.

In the FAT file system, only the dates and times of the last write can be modified. Creation and last-access dates and times are not affected.

The last-modification date and time will be changed if the extended attributes are modified.

Related Functions

- DosProtectClose
- DosProtectEnumAttribute
- DosProtectOpen
- DosProtectQueryFileInfo
- DosQueryPathInfo
- DosResetBuffer
- DosSetFileSize
- DosSetPathInfo

Example Code

This example creates a read-only file named DOSFDEL.DAT , then changes its attributes to normal file, and finally uses DosForceDelete to delete the file so that it cannot be restored using UNDELETE.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include os2.h
#include stdio.h

int main(VOID)

UCHAR       uchFileName   = "DOSFDEL.DAT";   /* File to delete    */
HFILE       fhDelFile       = 0;              /* File handle from DosOpenL  */
FILESTATUS3L fsts3FileInfo   = 0;  /* Information associated with file   */
ULONG       ulBufferSize    = sizeof(FILESTATUS3L); /* File info buffer size */
ULONG       ulOpenAction    = 0;                 /* Action taken by DosOpenL */
APIRET      rc              = NO_ERROR;          /* Return code             */
FHLOCK      FHLock          = 0;                 /* File handle lock        */

/* Create a read-only file */

rc = DosProtectOpenL(uchFileName, fhDelFile,
ulOpenAction, (LONGLONG)10, FILE_READONLY,
OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L, FHLock);
if (rc != NO_ERROR)
printf("DosProtectOpenL error return code = %u\n", rc);
return 1;


rc = DosProtectQueryFileInfo(fhDelFile, FIL_STANDARDL,
fsts3FileInfo, ulBufferSize, FHLock);   /* Get standard info */
if (rc != NO_ERROR)
printf("DosProtectQueryFileInfo error return code = %u\n", rc);
return 1;
 else  printf("File %s created read-only.\n",uchFileName);

/*   Change the file attributes to be "normal"  */

fsts3FileInfo.attrFile  = FILE_NORMAL;
rc = DosProtectSetFileInfo(fhDelFile, FIL_STANDARDL,
fsts3FileInfo, ulBufferSize, FHLock);
if (rc != NO_ERROR)
printf("DosProtectSetFileInfo error return code = %u\n", rc);
return 1;


rc = DosProtectClose(fhDelFile, FHLock);
/* Should verify that (rc != NO_ERROR) here... *//* Delete the file */

rc = DosForceDelete(uchFileName);
if (rc != NO_ERROR)
printf("DosForceDelete error return code = %u\n", rc);
return 1;
 else
printf("File %s has been deleted.\n",uchFileName);
  /* endif */

return NO_ERROR;
```

DosProtectSetFileLocksL

Purpose

DosProtectSetFileLocksL locks and unlocks a range of an open file.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosProtectSetFileLocksL **(HFILE hFile, PFILELOCKL pflUnlock, PFILELOCKL pflLock, ULONG timeout, ULONG flags, FHLOCK fhFileHandleLockID)**

Parameters

hFile HFILE)   input

File handle.

pflUnlock PFILELOCKL)   input

Address of the structure containing the offset and length of a range to be unlocked.

The structure is shown in the following figure

```
typedef struct FILELOCKL
LONGLONG       lOffset
LONGLONG       lRange
 FILELOCKL
```

pflLock PFILELOCKL)   input

Address of the structure containing the offset and length of a range to be locked

timeout ULONG)   input

The maximum time that the process is to wait for the requested locks.

The time is represented in milliseconds.

flags ULONG)   input

Flags that describe the action to be taken.

Possible actions are

Bits	Description
31 2	Reserved flags
1	Atomic This bit defines a request for atomic locking. If this bit is set to 1 and the lock range is equal to the unlock range, an atomic lock occurs. If this bit is set to 1 and the lock range is not equal to the unlock range, an error is returned. If this bit is set to 0, then the lock may or may not occur atomically with the unlock.
0	Share This bit defines the type of access that other processes may have to the file range that is being locked. If this bit is set to 0 (the default), other processes have no access to the locked file range. The current process has exclusive access to the locked file range, which must not overlap any other locked file range. If this bit is set to 1, the current process and other processes have shared read only access to the locked file range. A file range with shared access may overlap any other file range with shared access, but must not overlap any other file range with exclusive access.

fhFileHandleLockID FHLOCK)   input

The filehandle lockid returned by a previous DosProtectOpenL.

Returns

ulrc APIRET)   returns

Return Code.

DosProtectSetFileLocksL returns one of the following values

0	NO_ERROR
6	ERROR_INVALID_HANDLE

33	ERROR_LOCK_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER
95	ERROR_INTERRUPT
174	ERROR_ATOMIC_LOCK_NOT_SUPPORTED
175	ERROR_READ_LOCKS_NOT_SUPPORTED

Remarks

DosProtectSetFileLocksL allows a process to lock and unlock a range in a file. The time during which a file range is locked should be short.

If the lock and unlock ranges are both zero, ERROR_LOCK_VIOLATION is returned to the caller.

If you only want to lock a file range, set the unlock file offset and the unlock range length to zero.

If you only want to unlock a file range, set the lock file offset and the lock range length to zero.

When the Atomic bit of *flags* is set to 0, and DosProtectSetFileLocksL specifies a lock operation and an unlock operation, the unlock operation occurs first, and then the lock operation is performed. If an error occurs during the unlock operation, an error code is returned and the lock operation is not performed. If an error occurs during the lock operation, an error code is returned and the unlock remains in effect if it was successful.

The lock operation is atomic when all of these conditions are met

- The Atomic bit is set to 1 in *flags*
- The unlock range is the same as the lock range
- The process has shared access to the file range, and has requested exclusive access to it; or the process has exclusive access to the file range, and has requested shared access to it.

Some file system drivers (FSDs) may not support atomic lock operations. Versions of the operating system prior to OS/2 Version 2.00 do not support atomic lock operations. If the application receives the error code ERROR_ATOMIC_LOCK_NOT_SUPPORTED, the application should unlock the file range and then lock it using a non-atomic operation (with the atomic bit set to 0 in *flags*). The application should also refresh its internal buffers before making any changes to the file.

If you issue DosProtectClose to close a file with locks still in effect, the locks are released in no defined sequence.

If you end a process with a file open, and you have locks in effect in that file, the file is closed and the locks are released in no defined sequence.

The locked range can be anywhere in the logical file. Locking beyond the end of the file is not an error. A file range to be locked exclusively must first be cleared of any locked file sub-ranges or overlapping locked file ranges.

If you repeat DosProtectSetFileLocksL for the same file handle and file range, then you duplicate access to the file range. Access to locked file ranges is not duplicated across DosExecPgm. The proper method of using locks is to attempt to lock the file range, and to examine the return value.

The following table shows the level of access granted when the accessed file range is locked with an exclusive lock or a shared lock. Owner refers to a process that owns the lock. Non-owner refers to a process that does not own the lock.

Action	Exclusive Lock	Shared Lock
Owner read	Success	Success
Non-owner read	Wait for unlock. Return error code after time-out.	Success
Owner write	Success	Wait for unlock. Return error code after time-out.
Non-owner write	Wait for unlock. Return error code after time-out.	Wait for unlock. Return error code after time-out.

If only locking is specified, DosProtectSetFileLocksL locks the specified file range using *pflLock* If the lock operation cannot be accomplished, an error is returned, and the file range is not locked.

After the lock request is processed, a file range can be unlocked using the *pflUnlock* parameter of another DosProtectSetFileLocksL request. If unlocking cannot be accomplished, an error is returned.

Instead of denying read/write access to an entire file by specifying access and sharing modes with DosProtectOpenL requests, a process attempts to lock only the range needed for read/write access and examines the error code returned.

Once a specified file range is locked exclusively, read and write access by another process is denied until the file range is unlocked. If both unlocking and locking are specified by DosProtectSetFileLocksL, the unlocking operation is performed first, then locking is done.

Related Functions

- DosCancelLockRequestL
- DosDupHandle
- DosExecPgm
- DosProtectOpenL

Example Code

This example opens or creates and opens a file named FLOCK.DAT in protected mode, and updates it using file locks.

```
#define INCL_DOSFILEMGR       /* File Manager values */
#define INCL_DOSERRORS        /* DOS Error values    */
#include os2.h
#include stdio.h
#include string.h

int main(VOID)

HFILE     FileHandle   = NULLHANDLE;  /* File handle */
ULONG     Action       = 0,           /* Action taken by DosOpenL */
Wrote        = 0,           /* Number of bytes written by DosWrite */
i            = 0;           /* Loop index */
CHAR      FileData40 = "Forty bytes of demonstration text data\r\n";
APIRET    rc           = NO_ERROR;    /* Return code */
FHLOCK    FHLock       = 0;           /* File handle lock   */
FILELOCKL  LockArea     = 0,         /* Area of file to lock */
UnlockArea   = 0;         /* Area of file to unlock */

rc = DosProtectOpenL("flock.dat",                   /* File to open */
FileHandle,                   /* File handle */
Action,                       /* Action taken */
(LONGLONG)4000,                /* File primary allocation */
FILE_ARCHIVED,                 /* File attributes */
FILE_OPEN | FILE_CREATE,       /* Open function type */
OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
0L,                            /* No extended attributes */
FHLock);                      /* File handle lock */
if (rc != NO_ERROR)                        /* If open failed */
printf("DosProtectOpenL error return code = %u\n", rc);
return 1;


LockArea.lOffset = 0;              /* Start locking at beginning of file */
LockArea.lRange =  40;             /* Use a lock range of 40 bytes       */
UnLockArea.lOffset = 0;              /* Start unlocking at beginning of file */
UnLockArea.lRange =  0;              /* Use a unlock range of 0 bytes       */

/* Write 8000 bytes to the file, 40 bytes at a time */

for (i=0; i200; ++i)

rc = DosProtectSetFileLocksL(FileHandle,        /* File handle   */
UnlockArea,       /* Unlock previous record (if any) */
LockArea,         /* Lock current record */
2000L,             /* Lock time-out value of 2 seconds */
0L,                /* Exclusive lock, not atomic */
FHLock);           /* File handle lock */
if (rc != NO_ERROR)
printf("DosProtectSetFileLocksL error return code = %u\n", rc);
return 1;

rc = DosProtectWrite(FileHandle, FileData, sizeof(FileData), Wrote, FHLock);
if (rc != NO_ERROR)
printf("DosProtectWrite error return code = %u\n", rc);
return 1;

UnlockArea = LockArea;      /* Will unlock this record on next iteration */
LockArea.lOffset += 40;     /* Prepare to lock next record               */

 /* endfor - 8000 bytes written */
```

```
rc = DosProtectClose(FileHandle,FHLock);    /* Close file, release any locks */
/* Should check if (rc != NO_ERROR) here .... */

return NO_ERROR;
```

DosProtectSetFilePtrL

Purpose

DosProtectSetFilePtrL moves the read or write pointer according to the type of move specified.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosProtectSetFilePtrL **(HFILE hFile, LONGLONG ib, ULONG method, PLONGLONG ibActual, FHLOCK fhFileHandleLockID)**

Parameters

hFile HFILE)   input

The handle returned by a previous DosOpenL function.

ib LONGLONG)   input

The signed distance (offset) to move, in bytes.

method LONG)   input

The method of moving.

This field specifies the location in the file at which the read/write pointer starts before adding the *ib* offset. The values and their meanings are as shown in the following list

0	FILE_BEGIN Move the pointer from the beginning of the file.
1	FILE_CURRENT Move the pointer from the current location of the read/write pointer.
2	FILE_END Move the pointer from the end of the file. Use this method to determine a file s size.

ibActual PLONGLONG)   output

Address of the new pointer location.

fhFileHandleLockID FHLOCK)   input

The filehandle lockid returned by a previous DosProtectOpenL.

Returns

ulrc APIRET)   returns

Return Code.

DosProtectSetFilePtrL returns one of the following values

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
132	ERROR_SEEK_ON_DEVICE

131	ERROR_NEGATIVE_SEEK
130	ERROR_DIRECT_ACCESS_HANDLE

Remarks

The read/write pointer in a file is a signed 64-bit number. A negative value for *ib* moves the pointer backward in the file; a positive value moves it forward. DosProtectSetFilePtrL cannot be used to move to a negative position in the file.

DosProtectSetFilePtrL cannot be used for a character device or pipe.

Related Functions

- DosProtectOpenL
- DosProtectRead
- DosProtectSetFileSizeL
- DosProtectWrite

Example Code

This example opens or creates and opens a file named DOSPROT.DAT , writes a string to it, returns the file pointer to the beginning of the file, reads it, and finally closes it using DosProtect functions.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include os2.h
#include stdio.h
#include string.h

int main(VOID)
HFILE  hfFileHandle   = 0L;
ULONG  ulAction       = 0;
ULONG  ulBytesRead    = 0;
ULONG  ulWrote        = 0;
LONGLONG  ullLocal    = 0;
UCHAR  uchFileName20  = "dosprot.dat",
uchFileData100 = " ";
FHLOCK FileHandleLock = 0;        /* File handle lock   */
APIRET rc             = NO_ERROR; /* Return code */

/* Open the file test.dat.  Make it read/write, open it */
/* if it already exists and create it if it is new.     */
rc = DosProtectOpenL(uchFileName,             /* File path name          */
hfFileHandle,                  /* File handle             */
ulAction,                      /* Action taken            */
(LONGLONG)100,                           /* File primary allocation */
FILE_ARCHIVED | FILE_NORMAL,    /* File attribute          */
OPEN_ACTION_CREATE_IF_NEW |
OPEN_ACTION_OPEN_IF_EXISTS,     /* Open function type      */
OPEN_FLAGS_NOINHERIT |
OPEN_SHARE_DENYNONE  |
OPEN_ACCESS_READWRITE,          /* Open mode of the file   */
0L,                             /* No extended attribute   */
FileHandleLock);               /* File handle lock id     */
if (rc != NO_ERROR)
printf("DosProtectOpenL error return code = %u\n", rc);
return 1;
 else
printf ("DosProtectOpenL Action taken = %u\n", ulAction);
 /* endif */

/* Write a string to the file */
strcpy (uchFileData, "testing...\n3...\n2...\n1\n");

rc = DosProtectWrite (hfFileHandle,       /* File handle                  */
(PVOID) uchFileData,       /* String to be written         */
sizeof (uchFileData),      /* Size of string to be written */
ulWrote,                 /* Bytes actually written       */
FileHandleLock);           /* File handle lock id          */if (rc != NO_ERROR)
printf("DosProtectWrite error return code = %u\n", rc);
return 1;
 else
printf ("DosProtectWrite Bytes written = %u\n", ulWrote);
 /* endif */

/* Move the file pointer back to the beginning of the file */
```

```
rc = DosProtectSetFilePtrL (hfFileHandle,   /* File Handle          */
(LONGLONG)0,            /* Offset               */
FILE_BEGIN,             /* Move from BOF        */
ullLocal,              /* New location address */
FileHandleLock);        /* File handle lock id  */
if (rc != NO_ERROR)
printf("DosSetFilePtr error return code = %u\n", rc);
return 1;


/* Read the first 100 bytes of the file */
rc = DosProtectRead (hfFileHandle,         /* File Handle                 */
uchFileData,                 /* String to be read           */
100L,                        /* Length of string to be read */
ulBytesRead,                /* Bytes actually read         */
FileHandleLock);             /* File handle lock id         */
if (rc != NO_ERROR)
printf("DosProtectRead error return code = %u\n", rc);
return 1;
 else
printf("DosProtectRead Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
 /* endif */

rc = DosProtectClose(hfFileHandle, FileHandleLock);   /* Close the file */
if (rc != NO_ERROR)
printf("DosProtectClose error return code = %u\n", rc);
return 1;

return NO_ERROR;
```

DosProtectSetFileSizeL

Purpose

DosProtectSetFileSizeL changes the size of a file.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosProtectSetFileSizeL **(HFILE hFile, LONGLONG cbSize, FHLOCK fhFileHandleLockID)**

Parameters

hFile HFILE)   input
Handle of the file whose size to be changed.

cbSize LONGLONG)   input
New size, in bytes, of the file.

fhFileHandleLockID FHLOCK)   input
The filehandle lockid obtained from DosProtectOpenL.

Returns

ulrc APIRET)   returns
Return Code.

DosProtectSetFileSizeL returns one of the following values

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
26	ERROR_NOT_DOS_DISK

33	ERROR_LOCK_VIOLATION
87	ERROR_INVALID_PARAMETER
112	ERROR_DISK_FULL

Remarks

When DosProtectSetFileSizeL is issued, the file must be open in a mode that allows write access.

The size of the open file can be truncated or extended. If the file size is being extended, the file system tries to allocate additional bytes in a contiguous (or nearly contiguous) space on the medium. The values of the new bytes are undefined.

Related Functions

- DosProtectOpenL
- DosProtectQueryFileInfo
- DosQueryPathInfo

Example Code

This example writes to a file named DOSPMAN.DAT , resets the buffer, and changes the size of the file using DosProtect functions.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include os2.h
#include stdio.h
#include string.h

int main(VOID)
HFILE  hfFileHandle   = 0L;     /* Handle for file being manipulated */
ULONG  ulAction       = 0;      /* Action taken by DosOpenL */
FHLOCK FileHandleLock = 0;      /* File handle lock   */

ULONG  ulWrote        = 0;      /* Number of bytes written by DosWrite */
UCHAR  uchFileName20  = "dospman.dat",     /* Name of file */
uchFileData4   = "DATA";            /* Data to write to file */
APIRET rc             = NO_ERROR;            /* Return code */

/* Open the file dosman.dat.  Use an existing file or create a new */
/* one if it doesn't exist.                                       */
rc = DosProtectOpenL(uchFileName, hfFileHandle, ulAction, (LONGLONG)4,
FILE_ARCHIVED | FILE_NORMAL,
OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
OPEN_FLAGS_NOINHERIT | OPEN_SHARE_DENYNONE  |
OPEN_ACCESS_READWRITE, 0L, FileHandleLock);
if (rc != NO_ERROR)
printf("DosProtectOpenL error return code = %u\n", rc);
return 1;


rc = DosProtectWrite (hfFileHandle, (PVOID) uchFileData,
sizeof (uchFileData), ulWrote, FileHandleLock);
if (rc != NO_ERROR)
printf("DosProtectWrite error return code = %u\n", rc);
return 1;


rc = DosResetBuffer (hfFileHandle);
if (rc != NO_ERROR)
printf("DosResetBuffer error return code = %u\n", rc);
return 1;
 /* endif */

rc = DosProtectSetFileSizeL (hfFileHandle, (LONGLONG)8, FileHandleLock);
if (rc != NO_ERROR)
printf("DosProtectSetFileSizeL error return code = %u\n", rc);
return 1;


return NO_ERROR;
```

DosQueryABIOSSupport

Purpose

DosQueryABIOSSupport returns flags that indicate various basic hardware configurations.

Syntax

```
#define INCL_DOSMODULEMGR
#include os2.h>
```

APIRET APRENTRY DosQueryABIOSSupport **(ULONG Reserved)**

Parameters

reserved (ULONG) input
Must be set to 0L. No other value is defined.

The following flags are returned

HW_CFG_MCA	0x01
HW_CFG_EISA	0x02
HW_CFG_ABIOS_SUPPORTED	0x04
HW_CFG_ABIOS_PRESENT	0x08
HW_CFG_PCI	0x10
HW_CFG_OEM_ABIOS	0x20
HW_CFG_IBM_ABIOS	0000
HW_CFG_PENTIUM_CPU	0x40

Example Code

```
int main(int argc, char *argv[], char *envp[])
{
   APIRET flags;

   flags = DosQueryABIOSSupport(0L);

   printf("H/W config %08x\n",flags);

   if (flags  HW_CFG_MCA)             printf("            0x01 => MCA Bus\n");
   if (flags  HW_CFG_EISA)            printf("            0x02 => EISA Bus\n");
   if (flags  HW_CFG_ABIOS_SUPPORTED) printf("            0x04 => ABIOS Supported\n");
   if (flags  HW_CFG_ABIOS_PRESENT)   printf("            0x08 => ABIOS Present\n");
   if (flags  HW_CFG_PCI)             printf("            0x10 => PCI Bus\n");
   if (flags  HW_CFG_OEM_ABIOS)       printf("            0x20 => OEM ABIOS\n");
   if (flags  HW_CFG_PENTIUM_CPU)     printf("            0x40 => Pentium or Higher CPU\n");

   return 0;
}
```

DosQueryFileInfo

Purpose

DosQueryFileInfo gets file information.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosQueryFileInfo **(HFILE hf, ULONG ulInfoLevel, PVOID pInfo, ULONG cbInfoBuf)**

Parameters

hf HFILE) input

The file handle.

ulInfoLevel ULONG) input

Level of file information required.

Specify a value

1	FIL_STANDARD
	Level 1 file information
11	FIL_STANDARDL
	Level 11 file information
2	FIL_QUERYEASIZE
	Level 2 file information
12	FIL_QUERYEASIZEL
	Level 12 file information
3	FIL_QUERYEASFROMLIST
	Level 3 file information

Level 4 is reserved.

The structures described in *pInfo* indicate the information returned for each of these levels.

pInfo PVOID) output

Address of the storage area where the system returns the requested level of file information.

File information, where applicable, is at least as accurate as the most recent DosClose, DosResetBuffer, DosSetFileInfo, or DosSetPathInfo.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)

pInfo contains the FILESTATUS3 data structure, in which file information is returned.

Level 11 File Information (*ulInfoLevel* == FIL_STANDARDL)

pInfo contains the FILESTATUS3L data structure, in which file information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)

pInfo contains the FILESTATUS4 data structure. This is similar to the Level 1 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an unsigned ULONG. On output, this field contains the size, in bytes, of the file s entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel* . The buffer size is less than or equal to twice the size of the file s entire EA set on disk.

Level 12 File Information (*ulInfoLevel* == FIL_QUERYEASIZEL)

pInfo contains the FILESTATUS4L data structure. This is similar to the Level 1 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an unsigned ULONG. On output, this field contains the size, in bytes, of the file s entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel* . The buffer size is less than or equal to twice the size of the file s entire EA set on disk.

Level 3 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLIST)

Input	*pInfo* contains an EAOP2 data structure. *fpGEA2List* points to a

GEA2 list defining the attribute names whose values are returned. The GEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the GEA2 list. The *oNextEntryOffset* field in the last entry of the GEA2 list must be 0. *fpFEA2List* points to a data area where the relevant FEA2 list is returned. The length field of this FEA2 list is valid, giving the size of the FEA2 list buffer. *oError* is ignored.

Output	*pInfo* is unchanged. The buffer pointed to by *fpFEA2List* is filled with the returned information. If the buffer that *fpFEA2List* points to is not large enough to hold the returned information (the return code is ERROR_BUFFER_OVERFLOW), *cbList* is still valid, assuming there is at least enough space for it. Its value is the size of the entire EA set on disk for the file, even though only a subset of attributes was requested.

cbInfoBuf ULONG) input
The length, in bytes, of *pInfo* .

Returns

ulrc APIRET) returns
Return Code.

DosQueryFileInfo returns one of the following values

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

Remarks

Information levels designated L should be used in order to get complete size information for files larger than 2GB. If information levels designated L are not used, a size of one byte will be returned for files which are >2GB in size.

In the FAT file system, only date and time information contained in Level 1 file information can be modified. Zero is returned for the creation and access dates and times.

To return information contained in any of the file information levels, DosQueryFileInfo must be able to read the open file. DosQueryFileInfo works only when the file is opened for read access, with a deny-write sharing mode specified for access by other processes. If another process that has specified conflicting sharing and access modes has already opened the file, any call to DosOpenL will fail.

DosEnumAttribute returns the lengths of EAs. This information can be used to calculate what size *pInfo* needs to be to hold full-extended-attribute (FEA) information returned by DosQueryFileInfo when Level 3 is specified. The size of the buffer is calculated as follows

Four bytes (for *oNextEntryOffset*) +
One byte (for *fEA* +
One byte (for *cbName*) +
Two bytes (for *cbValue*) +
Value of *cbName* (for the name of the EA) +
One byte (for terminating NULL in *cbName*) +
Value of *cbValue* (for the value of the EA)

Related Functions

- DosClose
- DosEnumAttribute
- DosOpen

- DosOpenL
- DosQueryPathInfo
- DosResetBuffer
- DosSetFileInfo
- DosSetFileSize
- DosSetFileSizeL
- DosSetPathInfo

Example Code

This example obtains the information of the CONFIG.SYS file.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include os2.h
#include stdio.h

int main(VOID)
UCHAR        uchFileName80 = "C\\CONFIG.SYS";  /* File to manipulate    */
FILESTATUS3L  fsts3ConfigInfo = 0;       /* Buffer for file information */
ULONG        ulBufSize     = sizeof(FILESTATUS3L);  /* Size of above buffer */
HFILE        hfConfig      = 0;             /* Handle for Config file      */
ULONG        ulOpenAction  = 0;             /* Action taken by DosOpenL     */
APIRET       rc            = NO_ERROR;      /* Return code                 */

rc = DosOpenL(uchFileName,                 /* File to open (path and name) */
hfConfig,                /* File handle returned         */
ulOpenAction,               /* Action taken by DosOpenL      */
(LONGLONG)0,0L,        /* Primary allocation and attributes (ignored) */
OPEN_ACTION_FAIL_IF_NEW |
OPEN_ACTION_OPEN_IF_EXISTS,  /* Open an existing file only   */
OPEN_FLAGS_NOINHERIT | OPEN_ACCESS_READONLY |
OPEN_SHARE_DENYNONE,         /* Read access only             */
0L);                         /* Extended attributes (ignored)*/

if (rc != NO_ERROR)
printf("DosOpenL error return code = %u\n", rc);
return 1;


rc = DosQueryFileInfo(hfConfig,   /* Handle of file                  */
FIL_STANDARDL,  /* Request standard (Level 11) info */
fsts3ConfigInfo, /* Buffer for file information  */
ulBufSize);    /* Size of buffer                  */
if (rc != NO_ERROR)
printf("DosQueryFileInfo error return code = %u\n", rc);
return 1;


rc = DosClose(hfConfig);      /* Close the file  (check RC in real life) */
printf("%s ---  File size %lld bytes\n",uchFileName, fsts3ConfigInfo.cbFile);
printf("Last updated %d/%d/%d; %d%2.2d\n",
fsts3ConfigInfo.fdateLastWrite.month,        /* Month            */
fsts3ConfigInfo.fdateLastWrite.day,          /* Day              */
(fsts3ConfigInfo.fdateLastWrite.year+1980L), /* Years since 1980 */
fsts3ConfigInfo.ftimeLastWrite.hours,        /* Hours            */
fsts3ConfigInfo.ftimeLastWrite.minutes);     /* Minutes          */

return NO_ERROR;
```

DosQueryMemState

Purpose

DosQueryMemState gets the status of a range of pages in memory. Its input parameters are an address and size. The address is rounded down to page boundary and size is rounded up to a whole number of pages. The status of the pages in the range is returned in the state parameter, and the size of the range queried is returned in the size parameter. If the pages in the range have conflicting states, then the state of the first page is returned.

Syntax

```
#define INCL_PROFILE
#include os2.h>
```

APIRET APIENTRY DosQueryMemState **(PVOID pMem, PULONG size, PULONG state)**

Parameters

pMem (PVOID)   input

size (PULONG)   input/output

state (PLONG)   output

Flags indicate the following page states

PAG_NPOUT 0x00000000	Page is not present, not in core.
PAG_PRESENT 0x00000001	Page is present.
PAG_NPIN 0x00000002	Page is not present, in core.
PAG_PRESMASK 0x00000003	Present state mask.
PAG_INVALID 0x00000000	Page is invalid.
PAG_RESIDENT 0x00000010	Page is resident.
PAG_SWAPPABLE 0x00000020	Page is swappable.
PAG_DISCARDABLE 0x00000030	Page is discardable.
PAG_TYPEMASK 0x00000030	Typemask.

Returns

ulrc (APIRET)   returns

Return Code.

DosQueryMemState returns one of the following values

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
487	ERROR_INVALID_ADDRESS

Related Functions

- DosQueryMem

Example Code

```
int main(int argc, char *argv[], char *envp[])
{
   APIRET rc=0;
   PVOID pMem;
   ULONG status;
   ULONG size;
   ULONG pages;
   ULONG onepage = 0x1000;

   if (argc  3) {

      printf("Syntax MEMSTATE address> size>\n");
      return 0;
   } else {

      pMem = (PVOID) strtoul(argv[1], NULL, 0);
      size = strtoul(argv[2], NULL, 0);
      pages = (size+0x0fff) >> 12;

      printf("address     state\n");
      while (pages--) {

         rc = DosQueryMemState(pMem, onepage, status);

         if (rc) printf("0x%08x DosQueryMemState returned %u\n",pMem, rc);
         else {
            printf("0x%08x 0x%08x ", pMem, status);
            if ((status  PAG_PRESMASK) == PAG_NPOUT) printf("not present, not in-core, ");
            else if (status  PAG_PRESENT) printf("present, in-core, ");
            else if (status  PAG_NPIN) printf("not present, in-core, ");

            if ((status  PAG_TYPEMASK) == PAG_INVALID) printf("invalid\n");
            if ((status  PAG_TYPEMASK) == PAG_RESIDENT) printf("resident\n");
            if ((status  PAG_TYPEMASK) == PAG_SWAPPABLE) printf("swappable\n");
            if ((status  PAG_TYPEMASK) == PAG_DISCARDABLE) printf("discardable\n");
         }
         pMem = (PVOID)((ULONG)pMem + 0x1000);

      } /* endwhile */

   } /* end if*/

   return rc;
}
```

Dos16QueryModFromCS

Purpose

Dos16QueryModFromCS queries the name, segment, and handle that corresponds to a 16 bit selector. It takes a selector as a parameter and returns information about the module (a protect mode application currently executing) owning that selector.

Syntax

```
#define INCL_DOSMODULEMGR
#include os2.h>
```

APIRET16 APIENTRY16 Dos16QueryModFromCS **(SEL sel, PQMRESULT qmresult)**

Parameters

sel (SEL)   input
Selector to be queried.

qmresult (QMRESULT)   output
Structure containing the queried information

```
typedef struct_QMRESULT{
USHORT seg;
USHORT htme;
char name[256];
} QMRESULT;

typedef QMRESULT*PQMRESULT;
```

Returns

ulrc (APIRET)   returns
Return Code.

Dos16QueryModFromCS returns one of the following values

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

Related Functions

- DosQueryModFromEIP
- DosSetExceptionHandler

Example Code

```
int main(int argc, char *argv[], char *envp[])
{
   SEL sel=0;
   QMRESULT qmresult;
   APIRET16 rc;

   if (argc!=2) {
      printf("QMODCS sel\n");
      return;
   } /* endif */

   sel=(SEL)strtoul(argv[1],NULL,16);

   rc=Dos16QueryModFromCS(sel, qmresult);

   if (rc != 0) {
      printf("DosQueryModFromCS returned rc=%u\n",rc);
      return rc;
   } /* endif */

   printf("Sel=0x%04x Handle=0x%04x Segment=0x%04x %s\n",
           sel,qmresult.hmte,qmresult.seg,qmresult.name);

   return 0;
}
```

DosQueryModFromEIP

Purpose

DosQueryModFromEIP queries a module handle and name from a given flat address. It takes a flat 32 bit address as a parameter and returns information about the module (a protect mode application currently executing) owning the storage.

Syntax

```
#define INCL_DOSMODULEMGR
#include os2.h>
```

APIRET APIENTRY DosQueryModFromEIP **(HMODULE *phMod, ULONG *pObjNum, ULONG BuffLen, PCHAR pBuff, ULONG *pOffset, ULONG Address)**

Parameters

phMod (PHMODULE) output
Address of a location in which the module handle is returned.

pObjNum (PULONG) output
Address of a ULONG where the module object number corresponding to the Address is returned. The object is zero based.

BuffLen (ULONG) input
Length of the user supplied buffer pointed to by pBuff.

pBuff (PCHAR)   output

Address of a user supplied buffer in which the module name is returned.

pOffset (PULONG)   output

Address of a where the offset to the object corresponding to the Address is returned. The offset is zero based.

Address (ULONG)   input

Input address to be queried.

Returns

ulrc (APIRET)   returns

Return Code.

DosQueryModFromEIP returns one of the following values

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
487	ERROR_INVALID_ADDRESS

Related Functions

- DosQueryModFromEIP
- DosSetExceptionHandler

Example Code

```
int main(int argc, char *argv[], char *envp[])
{
   HMODULE hMod;
   ULONG ObjNum;
   ULONG Offset;
   ULONG eip;
   APIRET rc;
   char Buff[256];


   if (argc !=2) {
      printf("QEIP \n");
      return 0;
   } /* endif */

   eip = strtoul(argv[1],NULL,0);

   rc=DosQueryModFromEIP( hMod,
                          ObjNum,
                          sizeof(Buff),
                          Buff,
                          Offset,
                          eip);
   if (rc!=0) {
      printf("DosQueryModFromEIP returned rc=%u\n",rc);
      return rc;
   } /* endif */

   printf("\nLinear Address 0x%08x\n",eip);
   printf("%s\n",Buff);
   printf("handle 0x%04x\n",hMod);
   printf("Object 0x%08x\n",ObjNum);
   printf("Offset 0x%08x\n",Offset);

   return 0;
}
```

DosQueryPathInfo

Purpose

DosQueryPathInfo gets file information for a file or subdirectory.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosQueryPathInfo **(PSZ pszPathName, ULONG ulInfoLevel, PVOID pInfoBuf, ULONG cbInfoBuf)**

Parameters

pszPathName PSZ) input
Address of the ASCIIZ file specification of the file or subdirectory.

Global file-name characters can be used in the name only for level 5 file information.

DosQuerySysInfo is called by an application during initialization to determine the maximum path length allowed by the operating system.

ulInfoLevel ULONG) input
The level of path information required.

Specify a value

1	FIL_STANDARD Level 1 file information
11	FIL_STANDARDL Level 11 file information
2	FIL_QUERYEASIZE Level 2 file information
12	FIL_QUERYEASIZEL Level 12 file information
3	FIL_QUERYEASFROMLIST Level 3 file information
5	FIL_QUERYFULLNAME Level 5 file information

Level 4 is reserved.

The structures described in *pInfoBuf* indicate the information returned for each of these levels.

pInfoBuf PVOID) output
Address of the storage area containing the requested level of path information.

Path information, where applicable, is based on the most recent DosClose, DosResetBuffer, DosSetFileInfo, or DosSetPathInfo.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfoBuf contains the FILESTATUS3 data structure, in which path information is returned.

Level 11 File Information (*ulInfoLevel* == FIL_STANDARDL)
pInfoBuf contains the FILESTATUS3L data structure, in which path information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfoBuf contains the FILESTATUS4 data structure. This is similar to the Level 1 structure, with the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an unsigned LONG On output, this field contains the size, in bytes, of the file s entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel* . The buffer size is less than or equal to twice the size of the file s entire EA set on disk.

Level 12 File Information (*ulInfoLevel* == FIL_QUERYEASIZEL)
pInfoBuf contains the FILESTATUS4L data structure. This is similar to the Level 1 structure, with

the addition of the *cbList* field after the *attrFile* field.

The *cbList* field is an unsigned ULONG On output, this field contains the size, in bytes, of the file s entire extended attribute (EA) set on disk. You can use this value to calculate the size of the buffer required to hold the EA information returned when a value of 3 is specified for *ulInfoLevel* . The buffer size is less than or equal to twice the size of the file s entire EA set on disk.

Level 3 File Information (*ulInfoLevel* == FIL_QUERYEASFROMLIST)
This is a subset of the EA information of the file.

Input

ulInfoLevel contains an EAOP2 data structure. *fpGEA2List* points to a GEA2 that defines the attribute names whose values are returned. The GEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the GEA2 list. The *oNextEntryOffset* field in the last entry of the GEA2 list must be zero. *fpFEA2List* points to a data area where the relevant FEA2 list is returned. The length field of this FEA2 list is valid, giving the size of the FEA2 list buffer. *oError* is ignored.

Output

pInfoBuf is unchanged. If an error occurs, *oError* points to the GEA2 entry that caused the error. The buffer pointed to by *fpFEA2List* is filled in with the returned information. If the buffer that *fpFEA2List* points to is not large enough to hold the returned information (the return code is ERROR_BUFFER_OVERFLOW), *cbList* is still valid, assuming there is at least enough space for it. Its value is the size, in bytes, of the file s entire EA set on disk, even though only a subset of attributes was requested. The size of the buffer required to hold the EA information is less than or equal to twice the size of the file s entire EA set on disk.

Level 5 File Information (*ulInfoLevel* == FIL_QUERYFULLNAME)
Level 5 returns the fully-qualified ASCIIZ name of *pszPathName* in *pInfoBuf* . *pszPathName* may contain global file-name characters.

cbInfoBuf ULONG) input
The length, in bytes, of *pInfoBuf* .

Returns

ulrc APIRET) returns
Return Code.

DosQueryPathInfo returns one of the following values

0	NO_ERROR
3	ERROR_PATH_NOT_FOUND
32	ERROR_SHARING_VIOLATION
111	ERROR_BUFFER_OVERFLOW
124	ERROR_INVALID_LEVEL
206	ERROR_FILENAME_EXCED_RANGE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

Remarks

In the FAT file system, only date and time information contained in Level 1 file information can be modified. Zero is returned for the creation and access dates and times.

For DosQueryPathInfo to return information contained in any of the file information levels, the file object must be opened for read access, with a deny-write sharing mode specified for access by other processes. Thus, if the file object is already accessed by another process that holds conflicting sharing and access rights, a call to DosQueryPathInfo fails.

Related Functions

- DosClose
- DosCreateDir

- DosEnumAttribute
- DosOpen
- DosOpenL
- DosQueryFileInfo
- DosResetBuffer
- DosSetFileInfo
- DosSetPathInfo

Example Code

The first example obtains information about the file STARTUP.CMD. The second example obtains information about the directory SYSTEM.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include os2.h
#include stdio.h

int main(VOID)
UCHAR        uchFileName80 = "C\\STARTUP.CMD";  /* File to manipulate    */
FILESTATUS3L  fsts3ConfigInfo = 0;       /* Buffer for file information */
ULONG        ulBufSize     = sizeof(FILESTATUS3L);  /* Size of above buffer */
APIRET       rc            = NO_ERROR;      /* Return code                 */

rc = DosQueryPathInfo(uchFileName,   /* Path and name of file           */
FIL_STANDARDL,  /* Request standard (Level 11) info */
fsts3ConfigInfo, /* Buffer for file information  */
ulBufSize);    /* Size of buffer                  */
if (rc != NO_ERROR)
printf("DosQueryPathInfo error return code = %u\n", rc);
return 1;


printf("%s ---  File size %lld bytes\n",uchFileName, fsts3ConfigInfo.cbFile);
printf("Last updated %d/%d/%d; %d%2.2d\n",
fsts3ConfigInfo.fdateLastWrite.month,        /* Month            */
fsts3ConfigInfo.fdateLastWrite.day,          /* Day              */
(fsts3ConfigInfo.fdateLastWrite.year+1980L), /* Years since 1980 */
fsts3ConfigInfo.ftimeLastWrite.hours,        /* Hours            */
fsts3ConfigInfo.ftimeLastWrite.minutes);     /* Minutes          */

return NO_ERROR;

#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include os2.h
#include stdio.h

int main(VOID)
UCHAR        uchPathName255 = "C\\OS2\\SYSTEM"; /* Path of interest     */
FILESTATUS3L  fsts3ConfigInfo = 0;       /* Buffer for path information */
ULONG        ulBufSize     = sizeof(FILESTATUS3L);  /* Size of above buffer */
APIRET       rc            = NO_ERROR;      /* Return code                 */

rc = DosQueryPathInfo(uchPathName,   /* Name of path                    */
FIL_STANDARDL,  /* Request standard (Level 11) info */
fsts3ConfigInfo, /* Buffer for information       */
ulBufSize);       /* Size of buffer               */
if (rc != NO_ERROR)
printf("DosQueryPathInfo error return code = %u\n", rc);
return 1;


printf("Information for subdirectory %s\n",uchPathName);
printf("Last updated %d/%d/%d; %d%2.2d\n",
fsts3ConfigInfo.fdateLastWrite.month,        /* Month            */
fsts3ConfigInfo.fdateLastWrite.day,          /* Day              */
(fsts3ConfigInfo.fdateLastWrite.year+1980L), /* Years since 1980 */
fsts3ConfigInfo.ftimeLastWrite.hours,        /* Hours            */
fsts3ConfigInfo.ftimeLastWrite.minutes);     /* Minutes          */

return NO_ERROR;
```

DosQuerySysInfo

Purpose

DosQuerySysInfo returns values of static system variables.

Syntax

```
#define INCLDOSMISC
#include  os2.h
```

APIRET DosQuerySysInfo **(ULONG iStart, ULONG iLast, PVOID pBuf, ULONG cbBuf)**

Parameters

iStart ULONG)   input
Ordinal of the first system variable to return.

iLast ULONG)   input
Ordinal of the last system variable to return.

pBuf PVOID)   output
Address of the data buffer where the system returns the variable values.

cbBuf ULONG)   input
Length, in bytes, of the data buffer.

Returns

ulrc APIRET)   returns
Return Code.

DosQuerySysInfo returns one of the following values

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW

Remarks

DosQuerySysInfo returns a single system variable or a range of system variables to a user-allocated buffer. To request a single system variable, set *iStart* equal to *iLast* . To request a range of system variables, set *iStart* less than *iLast* .

Each system variable is a ULONG value. The following list gives the ordinal index, name, and description of the system variables.

1 QSV_MAX_PATH_LENGTH

Maximum length, in bytes, of a path name.

2 QSV_MAX_TEXT_SESSIONS

Maximum number of text sessions.

3 QSV_MAX_PM_SESSIONS

Maximum number of PM sessions.

4 QSV_MAX_VDM_SESSIONS

Maximum number of DOS sessions.

5 QSV_BOOT_DRIVE

Drive from which the system was started (1 means drive A, 2 means drive B, and so on).

6 QSV_DYN_PRI_VARIATION

Dynamic priority variation flag (0 means absolute priority, 1 means dynamic priority).

7	QSV_MAX_WAIT
	Maximum wait in seconds.
8	QSV_MIN_SLICE
	Minimum time slice in milliseconds.
9	QSV_MAX_SLICE
	Maximum time slice in milliseconds.
10	QSV_PAGE_SIZE
	Memory page size in bytes. This value is 4096 for the 80386 processor.
11	QSV_VERSION_MAJOR
	Major version number (see note below).
12	QSV_VERSION_MINOR
	Minor version number (see note below).
13	QSV_VERSION_REVISION
	Revision number (see note below).
14	QSV_MS_COUNT
	Value of a 32-bit, free-running millisecond counter. This value is zero when the system is started.
15	QSV_TIME_LOW
	Low-order 32 bits of the time in seconds since January 1, 1970 at 0 00 00.
16	QSV_TIME_HIGH
	High-order 32 bits of the time in seconds since January 1, 1970 at 0 00 00.
17	QSV_TOTPHYSMEM
	Total number of bytes of physical memory in the system.
18	QSV_TOTRESMEM
	Total number of bytes of resident memory in the system.
19	QSV_TOTAVAILMEM
	Maximum number of bytes of memory that can be allocated by all processes in the system. This number is advisory and is not guaranteed, since system conditions change constantly.
20	QSV_MAXPRMEM
	Maximum number of bytes of memory that this process can allocate in its private arena. This number is advisory and is not guaranteed, since system conditions change constantly.
21	QSV_MAXSHMEM
	Maximum number of bytes of memory that a process can allocate in the shared arena. This number is advisory and is not guaranteed, since system conditions change constantly.
22	QSV_TIMER_INTERVAL
	Timer interval in tenths of a millisecond.
23	QSV_MAX_COMP_LENGTH
	Maximum length, in bytes, of one component in a path name.
24	QSV_FOREGROUND_FS_SESSION

Session ID of the current foreground full-screen session. Note that this only applies to full-screen sessions. The Presentation Manager session (which displays Vio-windowed, PM, and windowed DOS Sessions) is full-screen session ID 1.

25 QSV_FOREGROUND_PROCESS

Process ID of the current foreground process.

26 QSV_NUMPROCESSORS

Number of processors in the machine

27 QSV_MAXHPRMEM

Maximum amount of free space in process's high private arena. Because system conditions change constantly, this number is advisory and is not guaranteed. In addition, this number does not indicate the largest single memory object you can allocate because the arena may be fragmented.

28 QSV_MAXHSHMEM

Maximum amount of free space in process's high shared arena. Because system conditions change constantly, this number is advisory and is not guaranteed. In addition, this number does not indicate the largest single memory object you can allocate because the arena may be fragmented.

29 QSV_MAXPROCESSES

Maximum number of concurrent processes supported.

30 QSV_VIRTUALADDRESSLIMIT

Size of the user's address space in megabytes (that is, the value of the rounded VIRTUALADDRESSLIMIT)

30 QSV_MAX

Note: Major, minor and revision numbers for versions of OS/2 operating system are described below

	Major	Minor	Revision
OS/2 2.0	20	00	0
OS/2 2.1	20	10	0
OS/2 2.11	20	11	0
OS/2 3.0	20	30	0
OS/2 4.0	20	40	0

An application can specify file objects managed by an installable file system that supports long file names. Because some installable file systems support longer names than others, the application should issue DosQuerySysInfo upon initialization.

DosQuerySysInfo returns the maximum path length (QSV_MAX_PATH_LENGTH) supported by the installed file system. The path length includes the drive specifier (d), the leading backslash ( ), and the trailing null character. The value returned by DosQuerySysInfo can be used to allocate buffers for path names returned by other functions, for example, DosFindFirst and DosFindNext.

Related Functions

- DosCreateDir
- DosFindFirst
- DosFindNext
- DosOpen
- DosQueryCurrentDir
- DosQueryFSInfo
- DosQueryPathInfo
- DosSearchPath
- DosSetCurrentDir
- DosSetPathInfo
- DosSetFSInfo

Example Code

This example queries and displays the maximum length for a path name and the total amount of physical memory in bytes.

```
#define INCL_DOSMISC       /* DOS Miscellaneous values */
#define INCL_DOSERRORS     /* DOS Error values         */
#include os2.h
#include stdio.h

int main(VOID)

ULONG   aulSysInfoQSV_MAX = 0;       /* System Information Data Buffer */
APIRET  rc                  = NO_ERROR;  /* Return code                    */

rc = DosQuerySysInfo(1L,                 /* Request all available system   */
QSV_MAX,            /* information                    */
(PVOID)aulSysInfo,
sizeof(ULONG)*QSV_MAX);

if (rc != NO_ERROR)
printf("DosQuerySysInfo error return code = %u\n", rc);
return 1;
 else
printf("Maximum length for a path name is %u characters.\n",
aulSysInfoQSV_MAX_PATH_LENGTH-1);  /* Max length of path name */

printf("Total physical memory is %u bytes.\n",
aulSysInfoQSV_TOTPHYSMEM-1);       /* Total physical memory   */
 /* endif */

return NO_ERROR;
```

DosQuerySysState

Purpose

DosQuerySysState returns information about various resources in use by the system. The EntityList parameter determines which information is returned according to the bits set in this parameter.

Syntax

```
#define INCL_DOSPROFILE
#define INCL_DOSERRORS
#include os2.h>
```

APIRET APIENTRY DosQuerySysState **(ULONG EntityList, ULONG EntityLevel, PID pid, TID tid, PVOID pDataBuf, ULONG cbBuf)**

Parameters

EntityList (ULONG) input
Determines what information is returned. May be a combination of the following

QS_PROCESS	0x0001 Requests process information
QS_SEMAPHORE	0x0002 Requests semaphore information
QS_MTE	0x0004 Requests module information
QS_FILESYS	0x0008 Requests file system information
QS_SHMEMORY	0x0010 Requests shared memory information
QS_MODVER	0x0200 Requests module version information

EntityLevel (ULONG) input
Determines the extent of information returned for a given entity. This applies to QS_MTE entities only. If EntityLevel is also set to SQ_MTE, then module object information is returned.

pid (PID)   input

Restricts information to a particular process ID. If 0 is specified, then entities for all processes are returned.

tid (TID)   input

Restricts information to a particular thread ID. A value of zero only is supported, requesting all threads of a process.

pDataBuf (PVOID)   output

Pointer to the buffer allocated by the user into which entity structures are returned. If the buffer is of insufficient size, then an ERROR_BUFFER_OVERFLOW is returned.

cbBuf (ULONG)   input

Size of the buffer pointed to by pDataBuf in bytes.

Returns

ulrc (APIRET)   returns

Return Code.

DosQuerySysState returns one of the following values

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
111	ERROR_BUFFER_OVERFLOW
115	ERROR_PROTECTION_VIOLATION
124	ERROR_INVALID_LEVEL

Remarks

The information returned by DosQuerySysState begins with a pointer to the global record structure, qsGrec_s. Following this will be a series of other records which depend on what information was requested. Some of these subsequent record structures contain an identifier as their first member, which enables the returned information to be interpreted without any order being imposed.

Entities that may be requested are

Process information	QS_PROCESS
Semaphore information	QS_SEMAPHORE
Module information	QS_MTE
File system information	QS_FILESYS
Shared memory information	QS_SHMEMORY
Module Version information	QS_MODVER

Not all entities have been supported in earlier versions of OS/2.

The structures returned will be a combination of the following

qsGrec_t	Global Record structure
qsTrec_t	Thread Record structure
qsPrec_t	Process Record structure
qsS16rec_t	16 bit system semaphore structure
qsS16Headrec_t	16 bit system semaphore structure
qsMrec_t	Shared Memory Record structure
QSOPENQ	32 bit Open Semaphore structure
QSEVENT	32 bit Event Semaphore structure
QSMUTEX	32 bit Mutex semaphore structure
QSMUX	32 bit Mux semaphore structure
QSHUN	32 bit semaphore header structure
qsS32rec_t	32 bit semaphore header structure

qsLObjrec_t	Object level MTE information
qsLrec_t	System wide MTE information
qsExLrec_t	Module version information
qsSft_t	System wide FILE information one per open instance
qsFrec_t	System wide FILE information one per file name
qsPtrRec_t	System wide FILE information

Related Functions

- DosQueryMemState
- DosQuerySysInfo

Example Code

```
#define BUFSIZE 64*1024
int main(int argc, char *argv[], char *envp[])
{
   APIRET rc;
   qsGrec_t ** pBuf;
   qsGrec_t * pGrec;
   qsLrec_t * pLrec;


   pBuf=malloc(BUFSIZE); /* allocate a 64K buffer */
   if (pBuf == NULL) {
      printf("Not enough memoryan");
      return ERROR_NOT_ENOUGH_MEMORY;
   } /* endif */

   /* query module information */

   rc=DosQuerySysState(QS_MTE, 0L, 0L, 0L, pBuf, BUFSIZE);
   if (rc!=0) {
      printf("DosQuerySysState returned rc=%u\n",rc);
      return rc;
   } /* endif */

   pGrec = *  pBuf;

   printf("Threads=%u 32-bit Sems=%u File Names=%u\n\n",
                pGrec->cThrds,
                pGrec->c32SSem,
                pGrec->cMFTNodes);

   pLrec = (ULONG)pGrec + sizeof(qsGrec_t);

   while (pLrec) {
      if (pLrec->pName) printf("hmte=%04x %s\n", pLrec->hmte, pLrec->pName);
      pLrec = pLrec->pNextRec;
   } /* endwhile */

   return rc;
}
```

DosQueryThreadAffinity

Purpose

DosQueryThreadAffinity allows a thread to inquire for the current thread's processor affinity mask and the system's capable processor affinity mask.

Syntax

APIRET DosQueryThreadAffinity **(ULONG scope, PMPAffinity pAffinityMask)**

Parameters

scope(ULONG) input

AFNTY_THREAD	Return the current threads processor affinity mask.
AFNTY_SYSTEM	Return the system's current capable processor affinity mask.

pAffinityMask(PMPAffinity) input
Address of MPAffinity structure to receive the affinity mask. Processors 0 31 are in mask [0] and processors 32 63 are in mask [1].

Returns

ulrc APIRET) returns
Return Code.

DosQueryThreadAffinity returns one of the following values

13	ERROR_INVALID_DATA
87	ERROR_INVALID_PARAMETER

Remarks

DosQueryThreadAffinity allows a thread to ask the Processor Affinity Mask for

1. The current thread's processor affinity mask, scope =AFNTY_THREAD, returns qwThreadAffinity, for the calling thread.
2. The system's capable processor affinity mask, scope=AFNTY_SYSTEM, returns qwCapableAffinity for the system. The caller may then use any subset of the returned affinity mask to change the threads processor affinity in a later call to DosSetThreadAffinity.

Related Functions

- DosSetThreadAffinity

Example Code

```
#define INCL_DOS
#define INCL_32
#define INCL_DOSERRORS
#define INCL_NOPMAPI
#include os2.h>
#include stdio.h>

int main(void) {
APIRET rc;
MPAFFINITY affinity;

rc = DosQueryThreadAffinity(AFNTY_SYSTEM, affinity);
printf("Query system's affinity rc = %08.8xh\n",rc);
printf("Query system's affinity affinity[0] = %08.8xh, affinity[1] = %08.8xh\n",
        affinity.mask[0], affinity.mask[1]);
return rc;
}
```

DosRead

Purpose

DosRead reads the specified number of bytes from a disk to a buffered location.

Syntax

```
#define INCL_DOSFILEMGR
#include os2.h>
```

APIRET DosRead **(HFILE hf, PVOID pBuffer, ULONG cbRead, PULONG pcbActual)**

Parameters

hFile (HFILE) input
File handle obtained from DosOpen.

pBuffer (PVOID) output
Address of the buffer to receive the bytes read.

cbRead (ULONG) input
The number of bytes to be read into pBuffer. This must be a multiple of the sector size (512) for the raw file system.

pcbActual (PULONG) output
Address of the variable to receive the number of bytes actually read.

Returns

ulrc (APIRET) returns
Return Code.

DosRead returns one of the following values

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
26	ERROR_NOT_DOS_DISK
33	ERROR_LOCK_VIOLATION
109	ERROR_BROKEN_PIPE
234	ERROR_MORE_DATA

Remarks

DosRead begins reading from the current file pointer position. The file pointer is updated by reading the data.

The requested number of bytes might not be read. If the value returned in *pcbActual* is less than requested, the process tried to read past the end of the disk.

A value of zero for *cbRead* is not considered an error. It is treated as a null operation.

Using the raw file system on logical partitions requires you to lock and unlock the volume using the DosDevIOCtl Category 8, DSK_LOCKDRIVE and DSK_UNLOCKDRIVE. Reads and writes will not succeed until the logical drive is locked.

The raw file system requires that the number of bytes read be a multiple of the sector size (512).

Related Functions

- DosOpen
- DosListIO
- DosSetFilePtr
- DosWrite

Example Code

The following is NOT a complete usable program. It is simply intended to provide an idea of how to use Raw I/O File System APIs (e.g. DosOpen, DosRead, DosWrite, DosSetFilePtr, and DosClose).

This example opens physical disk #1 for reading and physical disk #2 for writing. DosSetFilePtr is used to set the pointer to the beginning of the disks. Using DosRead and DosWrite, 10 megabytes of data is transferred from disk #1 to disk #2. Finally, DosClosed is issued to close the disk handles.

It is assumed that the size of each of the two disks is at least 10 megabytes.

```
#define INCL_DOSFILEMGR          /* Include File Manager APIs */
#define INCL_DOSMEMMGR           /* Includes Memory Management APIs */
#define INCL_DOSERRORS           /* DOS Error values */
#include os2.h>
#include stdio.h>
#include string.h>

#define SIXTY_FOUR_K 0x10000
#define ONE_MEG     0x100000
#define TEN_MEG     10*ONE_MEG

#define UNC_DISK1  "\\\\.\\Physical_Disk1"
#define UNC_DISK2  "\\\\.\\Physical_Disk2"

int main(void) {
   HFILE  hfDisk1        = 0;      /* Handle for disk #1 */
   HFILE  hfDisk2        = 0;      /* Handle for disk #2 */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   ULONG  cbRead         = 0;      /* Bytes to read */
   ULONG  cbActualRead   = 0;      /* Bytes read by DosRead */
   ULONG  cbWrite        = 0;      /* Bytes to write */
   ULONG  ulLocation     = 0;
   ULONG  cbActualWrote  = 0;      /* Bytes written by DosWrite */
   UCHAR  uchFileName1[20]  = UNC_DISK1, /* UNC Name of disk 1 */
          uchFileName2[20]  = UNC_DISK2; /* UNC Name of disk 2 */
   PBYTE  pBuffer        = 0;
   ULONG  cbTotal        = 0;

   APIRET rc             = NO_ERROR;            /* Return code */

   /* Open a raw file system disk #1 for reading */
   rc = DosOpen(uchFileName1,               /* File name */
                hfDisk1,                   /* File handle */
                ulAction,                  /* Action taken by DosOpen */
                0L,                         /* no file size */
                FILE_NORMAL,                /* File attribute */
                OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
                OPEN_SHARE_DENYNONE |       /* Access mode */
                OPEN_ACCESS_READONLY,
                0L);                        /* No extented attributes */
   if (rc != NO_ERROR) {
      printf("DosOpen error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Set the pointer to the begining of the disk */
   rc = DosSetFilePtr(hfDisk1,      /* Handle for disk 1 */
                      0L,           /* Offset must be multiple of 512 */
                      FILE_BEGIN,   /* Begin of the disk */
                      ulLocation); /* New pointer location */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Open a raw file system disk #2 for writing */
   rc = DosOpen(uchFileName2,               /* File name */
                hfDisk2,                   /* File handle */
                ulAction,                  /* Action taken by DosOpen */
                0L,                         /* no file size */
                FILE_NORMAL,                /* File attribute */
                OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
                OPEN_SHARE_DENYNONE |       /* Access mode */
                OPEN_ACCESS_READWRITE,
                0L);                        /* No extented attributes */
   if (rc != NO_ERROR) {
      printf("DosOpen error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Set the pointer to the begining of the disk */
   rc = DosSetFilePtr(hfDisk2,      /* Handle for disk 1 */
                      0L,           /* Offset must be multiple of 512 */
                      FILE_BEGIN,   /* Begin of the disk */
                      ulLocation); /* New pointer location */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error rc = %u\n", rc);
      return(1);
   } /* endif */
   /* Allocate 64K of memory for transfer operations */
   rc = DosAllocMem((PPVOID)pBuffer, /* Pointer to buffer */
                     SIXTY_FOUR_K,      /* Buffer size */
                     PAG_COMMIT |     /* Allocation flags */
                     PAG_READ |
                     PAG_WRITE);
   if (rc != NO_ERROR) {
      printf("DosAllocMem error rc = %u\n", rc);
      return(1);
   } /* endif */

   cbRead = SIXTY_FOUR_K;
   while (rc == NO_ERROR  cbTotal  TEN_MEG) {

      /* Read from #1 */
      rc = DosRead(hfDisk1,         /* Handle for disk 1 */
                   pBuffer,         /* Pointer to buffer */
                   cbRead,          /* Size must be multiple of 512 */
                   cbActualRead);  /* Actual read by DosOpen */
      if (rc) {
         printf("DosRead error return code = %u\n", rc);
         return 1;
      }

      /* Write to disk #2 */
      cbWrite = cbActualRead;
      rc = DosWrite(hfDisk2,         /* Handle for disk 2 */
                    pBuffer,         /* Pointer to buffer */
                    cbWrite,         /* Size must be multiple of 512 */
                    cbActualWrote); /* Actual written by DosOpen */
      if (rc) {
         printf("DosWrite error return code = %u\n", rc);
         return 1;
      }
      if (cbActualRead != cbActualWrote) {
         printf("Bytes read (%u) does not equal bytes written (%u)\n",
                cbActualRead, cbActualWrote);
         return 1;
      }
      cbTotal += cbActualRead; /* Update total transferred */
   }

   printf("Transfer successfully %d bytes from disk #1 to disk #2.\n",
          cbTotal);

   /* Free allocated memmory */
   rc = DosFreeMem(pBuffer);
   if (rc != NO_ERROR) {
      printf("DosFreeMem error return code = %u\n", rc);
      return 1;
   }

   rc = DosClose(hfDisk1);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
   }

   rc = DosClose(hfDisk2);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
   }
return NO_ERROR;
}
```

DosReplaceModule

Purpose

DosReplaceModule replaces or caches a module that is in use.

Syntax

```
#define INCL_DOSMEMMGR
#include os2.h>
```

APIRET APIENTRY DosReplaceModule **(PSZ pszOldModule, PSZ pszNewModule, PSZ pszBackupModule)**

Parameters

pszOldModule (PSZ) input
: Points to the name of the existing module. Required.

pszNewModule (PSZ) input
: Points to the name of the new module. Optional.

pszBackupModule (PSZ) input
: Points to the name to be used for saving a copy of the old module. Optional.

Returns

ulrc (APIRET) returns
: Return Code.

DosReplaceModule returns one of the following values

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
5	ERROR_ACCESS_DENIED
17	ERROR_NOT_THE_SAME_DEVICE
26	ERROR_NOT_DOS_DISK
32	ERROR_SHARING VIOLATION
87	ERROR_INVALID_PARAMETER
108	ERROR_DRIVE_LOCKED
112	ERROR_DISK_FULL
267	ERROR_DIRECTORY
296	ERROR_MODULE_IN_USE
731	ERROR_MODULE_CORRUPTED

Remarks

When a DLL or EXE file is in use by the system, the file is locked. It cannot, therefore, be replaced on the harddisk by a newer copy. DosReplaceModule allows the replacement on the disk of the module while the system continues to run the old module. The contents of the old module file are cached in the swap file by the system and the load module file is closed. A backup copy of the file may be created for recovery purposes should the install program fail. If a backup module is not specified, then no backup will be made. The new module takes the place of the original module on the disk.

Note: The system will continue to use the cached old module until all references to it are released. The next reference to the module will cause a reload from the new module on disk. If a new module is not specified, then the old module file will be cached and the file closed.

Protect mode executable files only can be replaced by DosReplaceModule. It cannot be used for DOS/Windows(R) programs or for data files.

Related Functions

- DosLoadModule
- DosCopy

Example Code

```
int main(int argc, char *argv[], char *envp[])
```

```
{
   APIRET rc=0;
   PSZ pszOld;
   PSZ pszNew = NULL;
   PSZ pszBak = NULL;

   if (argc==1) {
      printf("REPMOD oldmod  \n");
      return rc;
   }

   if (argc>1) pszOld = argv[1];
   if (argc>2) pszNew = argv[2];
   if (argc>3) pszBak = argv[3];

   rc = DosReplaceModule(pszOld, pszNew, pszBak);

   if (rc) printf("DosReplaceModule returned %u\n",rc);

   else {
      if (argc==2) printf("%s successfully cached and closed\n", pszOld);
      else if (argc==3)
         printf("%s successfully cached and replaced with %s\n", pszOld, pszNew);
      else if (argc==4)
         printf("%s successfully copied to %s and replaced with %s\n", pszOld, pszBak, pszNew);
   }

   return rc;
}
```

DosSetFileInfo

Purpose

DosSetFileInfo sets file information.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosSetFileInfo **(HFILE hf, ULONG ulInfoLevel, PVOID pInfoBuf, ULONG cbInfoBuf)**

Parameters

hf HFILE)   input

File handle.

ulInfoLevel ULONG)   input

Level of file information being set.

Specify a value

1	FIL_STANDARD Level 1 file information
11	FIL_STANDARDL Level 11 file information
2	FIL_QUERYEASIZE Level 2 file information

The structures described in *pInfoBuf* indicate the information being set for each of these levels.

pInfoBuf PVOID)   input

Address of the storage area containing the structures for file information levels.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfoBuf contains the FILESTATUS3 data structure.

Level 11 File Information (*ulInfoLevel* == FIL_STANDARDL)
pInfo contains the FILESTATUS3L data structure, in which file information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfoBuf contains an EAOP2 data structure, and sets a series of EA name/value pairs.

Input	*pInfoBuf* is an EAOP2 data structure in which *fpFEA2List* points to a data area where the relevant FEA2LIST is to be found. *fpGEA2List* and *oError* are ignored.
Output	*fpGEA2List* and *fpFEA2List* are unchanged. The area pointed to by *fpFEA2List* is also unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 where the error occurred. The return code is the error code corresponding to the condition generating the error. If no error occurred, *oError* is undefined.

cbInfoBuf ULONG) input
The length, in bytes, of *pInfoBuf*.

Returns

ulrc APIRET) returns
Return Code.

DosSetFileInfo returns one of the following values

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
87	ERROR_INVALID_PARAMETER
122	ERROR_INSUFFICIENT_BUFFER
124	ERROR_INVALID_LEVEL
130	ERROR_DIRECT_ACCESS_HANDLE
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

Remarks

DosSetFileInfo is successful only when the file is opened for write access, and access by other processes is prevented by a deny-both sharing mode. If the file is already opened with conflicting sharing rights, any call to DosOpen will fail.

A value of 0 in the date and time components of a field does not change the field. For example, if both last write date and last write time are specified as 0 in the Level 1 information structure, then both attributes of the file are left unchanged. If either last write date or last write time are other than 0, both attributes of the file are set to the new values.

In the FAT file system, only the dates and times of the last write can be modified. Creation and last-access dates and times are not affected.

The last-modification date and time will be changed if the extended attributes are modified.

Related Functions

- DosClose
- DosEnumAttribute
- DosOpen
- DosOpenL
- DosQueryFileInfo

- DosQueryPathInfo
- DosResetBuffer
- DosSetFileSize
- DosSetFileSizeL
- DosSetPathInfo

Example Code

This example creates a read-only file named DOSFDEL.DAT , and then changes the file attributes. It uses DosForceDelete to delete the file so it cannot be restored using UNDELETE.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS error values    */
#include os2.h
#include stdio.h

int main(VOID)

UCHAR       uchFileName   = "DOSFDEL.DAT";   /* File we want to delete    */
HFILE       fhDelFile       = 0;               /* File handle from DosOpenL  */
FILESTATUS3L fsts3FileInfo   = 0;  /* Information associated with file   */
ULONG       ulBufferSize    = sizeof(FILESTATUS3L); /* File info buffer size */
ULONG       ulOpenAction    = 0;                 /* Action taken by DosOpenL */
APIRET      rc              = NO_ERROR;          /* Return code             */

/* Create a read-only file */

rc = DosOpenL(uchFileName, fhDelFile,
ulOpenAction, (LONGLONG)10, FILE_READONLY,
OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L);
if (rc != NO_ERROR)
printf("DosOpenL error return code = %u\n", rc);
return 1;

rc = DosQueryFileInfo(fhDelFile, FIL_STANDARDL,
fsts3FileInfo, ulBufferSize);  /* Get standard info */
if (rc != NO_ERROR)
printf("DosQueryFileInfo error return code = %u\n", rc);
return 1;
 else  printf("File %s created read-only.\n",uchFileName);

fsts3FileInfo.attrFile  = FILE_NORMAL;
rc = DosSetFileInfo(fhDelFile, FIL_STANDARDL,
fsts3FileInfo, ulBufferSize);
if (rc != NO_ERROR)
printf("DosSetFileInfo error return code = %u\n", rc);
return 1;


rc = DosClose(fhDelFile);
/* should check (rc != NO_ERROR) here... */

/* Delete the file */

rc = DosForceDelete(uchFileName);
if (rc != NO_ERROR)
printf("DosForceDelete error return code = %u\n", rc);
return 1;
 else
printf("File %s has been deleted.\n",uchFileName);
 /* endif */

return NO_ERROR;
```

DosSetFileLocksL

Purpose

DosSetFileLocksL locks and unlocks a range of an open file.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosSetFileLocksL **(HFILE hFile, PFILELOCKL pflUnlock, PFILELOCKL pflLock, ULONG timeout, ULONG flags)**

Parameters

hFile HFILE)   input

File handle.

pflUnlock PFILELOCKL)   input

Address of the structure containing the offset and length of a range to be unlocked.

pflLock PFILELOCKL)   input

Address of the structure containing the offset and length of a range to be locked.

timeout ULONG)   input

The maximum time, in milliseconds, that the process is to wait for the requested locks.

flags ULONG)   input

Flags that describe the action to be taken.

This parameter has the following bit fields

Bits	Description
31 2	Reserved flags
1	Atomic This bit defines a request for atomic locking. If this bit is set to 1 and the lock range is equal to the unlock range, an atomic lock occurs. If this bit is set to 1 and the lock range is not equal to the unlock range, an error is returned. If this bit is set to 0, then the lock may or may not occur atomically with the unlock.
0	Share This bit defines the type of access that other processes may have to the file range that is being locked. If this bit is set to 0 (the default), other processes have no access to the locked file range. The current process has exclusive access to the locked file range, which must not overlap any other locked file range. If this bit is set to 1, the current process and other processes have shared read only access to the locked file range. A file range with shared access may overlap any other file range with shared access, but must not overlap any other file range with exclusive access.

Returns

ulrc APIRET)   returns

Return Code.

DosSetFileLocksL returns one of the following values

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
33	ERROR_LOCK_VIOLATION
36	ERROR_SHARING_BUFFER_EXCEEDED
87	ERROR_INVALID_PARAMETER

95	ERROR_INTERRUPT
174	ERROR_ATOMIC_LOCK_NOT_SUPPORTED
175	ERROR_READ_LOCKS_NOT_SUPPORTED

Remarks

DosSetFileLocksL allows a process to lock and unlock a range in a file. The time during which a file range is locked should be short.

If the lock and unlock ranges are both zero, ERROR_LOCK_VIOLATION is returned to the caller.

If you only want to lock a file range, set the unlock file offset and the unlock range length to zero.

If you only want to unlock a file range, set the lock file offset and the lock range length to zero.

When the Atomic bit of *flags* is set to 0, and DosSetFileLocksL specifies a lock operation and an unlock operation, the unlock operation occurs first, and then the lock operation is performed. If an error occurs during the unlock operation, an error code is returned and the lock operation is not performed. If an error occurs during the lock operation, an error code is returned and the unlock remains in effect if it was successful.

The lock operation is atomic when all of these conditions are met

- The Atomic bit is set to 1 in *flags*
- The unlock range is the same as the lock range
- The process has shared access to the file range, and has requested exclusive access to it; or the process has exclusive access to the file range, and has requested shared access to it.

Some file system drivers (FSDs) may not support atomic lock operations. Versions of the operating system prior to OS/2 Version 2.00 do not support atomic lock operations. If the application receives the error code ERROR_ATOMIC_LOCK_NOT_SUPPORTED, the application should unlock the file range and then lock it using a non-atomic operation (with the atomic bit set to 0 in *flags*). The application should also refresh its internal buffers before making any changes to the file.

If you issue DosClose to close a file with locks still in effect, the locks are released in no defined sequence.

If you end a process with a file open, and you have locks in effect in that file, the file is closed and the locks are released in no defined sequence.

The locked range can be anywhere in the logical file. Locking beyond the end of the file is not an error. A file range to be locked exclusively must first be cleared of any locked file subranges or overlapping locked file ranges.

If you repeat DosSetFileLocksL for the same file handle and file range, then you duplicate access to the file range. Access to locked file ranges is not duplicated across DosExecPgm. The proper method of using locks is to attempt to lock the file range, and to examine the return value.

The following table shows the level of access granted when the accessed file range is locked with an exclusive lock or a shared lock. Owner refers to a process that owns the lock. Non-owner refers to a process that does not own the lock.

Action	Exclusive Lock	Shared Lock
Owner read	Success	Success
Nonowner read	Wait for unlock. Return error code after timeout.	Success
Owner write	Success	Wait for unlock. Return error code after timeout.
Nonowner write	Wait for unlock. Return error code after timeout.	Wait for unlock. Return error code after timeout.

If only locking is specified, DosSetFileLocksL locks the specified file range using *pflLock* . If the lock operation cannot be accomplished, an error is returned, and the file range is not locked.

After the lock request is processed, a file range can be unlocked using the *pflUnlock* parameter of another DosSetFileLocksL request. If unlocking cannot be accomplished, an error is returned.

Instead of denying read/write access to an entire file by specifying access and sharing modes with DosOpenL requests, a process attempts to lock only the range needed for read/write access and examines the error code returned.

Once a specified file range is locked exclusively, read and write access by another process is denied until the file range is unlocked. If both unlocking and locking are specified by DosSetFileLocksL, the unlocking operation is performed first, then locking is done.

Related Functions

- DosCancelLockRequestL
- DosDupHandle
- DosExecPgm
- DosOpenL

Example Code

This example opens or creates and opens a file named FLOCK.DAT, and updates it using file locks.

```
#define INCL_DOSFILEMGR       /* File Manager values */
#define INCL_DOSERRORS        /* DOS Error values    */
#include os2.h
#include stdio.h
#include string.h

int main(VOID)

HFILE     FileHandle   = NULLHANDLE;  /* File handle */
ULONG     Action       = 0,           /* Action taken by DosOpenL */
Wrote        = 0,           /* Number of bytes written by DosWrite */
i            = 0;           /* Loop index */
CHAR      FileData40 = "Forty bytes of demonstration text data\r\n";
APIRET    rc           = NO_ERROR;    /* Return code */
FILELOCKL  LockArea     = 0,         /* Area of file to lock */
UnlockArea   = 0;         /* Area of file to unlock */

rc = DosOpenL("flock.dat",                   /* File to open */
FileHandle,                  /* File handle */
Action,                      /* Action taken */
(LONGLONG)4000,               /* File primary allocation */
FILE_ARCHIVED,                /* File attributes */
FILE_OPEN | FILE_CREATE,      /* Open function type */
OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE,
0L);                          /* No extended attributes */
if (rc != NO_ERROR)                         /* If open failed */
printf("DosOpenL error return code = %u\n", rc);
return 1;

LockArea.lOffset = 0;              /* Start locking at beginning of file */
LockArea.lRange =  40;             /* Use a lock range of 40 bytes       */
UnLockArea.lOffset = 0;            /* Start unlocking at beginning of file */
UnLockArea.lRange =  0;            /* Use a unlock range of 0 bytes       */

/* Write 8000 bytes to the file, 40 bytes at a time */
for (i=0; i200; ++i)
rc = DosSetFileLocksL(FileHandle,        /* File handle   */
UnlockArea,       /* Unlock previous record (if any) */
LockArea,         /* Lock current record */
2000L,             /* Lock time-out value of 2 seconds */
0L);               /* Exclusive lock, not atomic */
if (rc != NO_ERROR)
printf("DosSetFileLocksL error return code = %u\n", rc);
return 1;


rc = DosWrite(FileHandle, FileData, sizeof(FileData), Wrote);
if (rc != NO_ERROR)
printf("DosWrite error return code = %u\n", rc);
return 1;


UnlockArea = LockArea;      /* Will unlock this record on next iteration */
LockArea.lOffset += 40;     /* Prepare to lock next record               */

 /* endfor - 8000 bytes written */
rc = DosClose(FileHandle);    /* Close file, this releases outstanding locks */
/* Should check if (rc != NO_ERROR) here ... */
return NO_ERROR;
```

DosSetFilePtr

Purpose

DosSetFilePtr moves the read write pointer according to the type of move specified.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosSetFilePtr **(HFILE hFile, LONG ib, ULONG method, PULONG ibActual)**

Parameters

hFile HFILE)   input

The handle returned by a previous DosOpen function.

ib LONG)   input

The signed distance (offset) to move the read/write pointer, in bytes. The raw file system requires that the offset be a multiple of the sector size (512).

method ULONG)   input

The method of moving.

Specifies a location in the file from where the *ib* to move the read/write pointer starts. The values and their meanings are described in the following list

0 FILE_BEGIN

Move the pointer from the beginning of the file.

1 FILE_CURRENT

Move the pointer from the current location of the read write pointer.

2 FILE_END

Move the pointer from the end of the file. Use this method to determine a file s size.

ibActual PULONG)   output

Address of the new pointer location.

Returns

ulrc APIRET)   returns

Return Code.

DosSetFilePtr returns one of the following values

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
25	ERROR_SEEK
130	ERROR_DIRECT_ACCESS_HANDLE
131	ERROR_NEGATIVE_SEEK
132	ERROR_SEEK_ON_DEVICE

Remarks

The read/write pointer in a file is a signed 32-bit number. A negative value for *ib* moves the pointer backward; a positive value moves it forward. The resulting pointer value cannot be negative or larger than the disk or an error will be returned. The signed 32-bit value of the read/write pointer limits the raw file system to the first 2 Gigabytes of a disk.

Related Functions

- DosOpen

- DosListIO
- DosRead
- DosWrite

Example Code

The following is NOT a complete usable program. It is simply intended to provide an idea of how to use Raw I/O File System APIs (e.g. DosOpen, DosRead, DosWrite, DosSetFilePtr, and DosClose).

This example opens physical disk #1 for reading and physical disk #2 for writing. DosSetFilePtr is used to set the pointer to the beginning of the disks. Using DosRead and DosWrite, 10 megabytes of data is transferred from disk #1 to disk #2. Finally, DosClosed is issued to close the disk handles.

It is assumed that the size of each of the two disks is at least 10 megabytes.

```
#define INCL_DOSFILEMGR          /* Include File Manager APIs */
#define INCL_DOSMEMMGR           /* Includes Memory Management APIs */
#define INCL_DOSERRORS           /* DOS Error values */
#include os2.h>
#include stdio.h>
#include string.h>

#define SIXTY_FOUR_K 0x10000
#define ONE_MEG     0x100000
#define TEN_MEG     10*ONE_MEG

#define UNC_DISK1  "\\\\.\\Physical_Disk1"
#define UNC_DISK2  "\\\\.\\Physical_Disk2"

int main(void) {
   HFILE  hfDisk1        = 0;      /* Handle for disk #1 */
   HFILE  hfDisk2        = 0;      /* Handle for disk #2 */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   ULONG  cbRead         = 0;      /* Bytes to read */
   ULONG  cbActualRead   = 0;      /* Bytes read by DosRead */
   ULONG  cbWrite        = 0;      /* Bytes to write */
   ULONG  ulLocation     = 0;
   ULONG  cbActualWrote  = 0;      /* Bytes written by DosWrite */
   UCHAR  uchFileName1[20]  = UNC_DISK1, /* UNC Name of disk 1 */
          uchFileName2[20]  = UNC_DISK2; /* UNC Name of disk 2 */
   PBYTE  pBuffer        = 0;
   ULONG  cbTotal        = 0;

   APIRET rc             = NO_ERROR;            /* Return code */

   /* Open a raw file system disk #1 for reading */
   rc = DosOpen(uchFileName1,               /* File name */
             hfDisk1,                    /* File handle */
             ulAction,                   /* Action taken by DosOpen */
             0L,                          /* no file size */
             FILE_NORMAL,                 /* File attribute */
             OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
             OPEN_SHARE_DENYNONE |       /* Access mode */
             OPEN_ACCESS_READONLY,
             0L);                         /* No extented attributes */
   if (rc != NO_ERROR) {
      printf("DosOpen error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Set the pointer to the begining of the disk */
   rc = DosSetFilePtr(hfDisk1,      /* Handle for disk 1 */
                   0L,           /* Offset must be multiple of 512 */
                   FILE_BEGIN,   /* Begin of the disk */
                   ulLocation); /* New pointer location */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Open a raw file system disk #2 for writing */
   rc = DosOpen(uchFileName2,               /* File name */
             hfDisk2,                    /* File handle */
             ulAction,                   /* Action taken by DosOpen */
             0L,                          /* no file size */
             FILE_NORMAL,                 /* File attribute */
             OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
             OPEN_SHARE_DENYNONE |       /* Access mode */
             OPEN_ACCESS_READWRITE,
             0L);                        /* No extented attributes */
   if (rc != NO_ERROR) {
      printf("DosOpen error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Set the pointer to the begining of the disk */
   rc = DosSetFilePtr(hfDisk2,      /* Handle for disk 1 */
                   0L,           /* Offset must be multiple of 512 */
                   FILE_BEGIN,   /* Begin of the disk */
                   ulLocation); /* New pointer location */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error rc = %u\n", rc);
      return(1);
   } /* endif */


   /* Allocate 64K of memory for transfer operations */
   rc = DosAllocMem((PPVOID)pBuffer, /* Pointer to buffer */
                     SIXTY_FOUR_K,      /* Buffer size */
                     PAG_COMMIT |     /* Allocation flags */
                     PAG_READ |
                     PAG_WRITE);
   if (rc != NO_ERROR) {
      printf("DosAllocMem error rc = %u\n", rc);
      return(1);
   } /* endif */

   cbRead = SIXTY_FOUR_K;
   while (rc == NO_ERROR  cbTotal  TEN_MEG) {

      /* Read from #1 */
      rc = DosRead(hfDisk1,         /* Handle for disk 1 */
                   pBuffer,         /* Pointer to buffer */
                   cbRead,          /* Size must be multiple of 512 */
                   cbActualRead);  /* Actual read by DosOpen */
      if (rc) {
         printf("DosRead error return code = %u\n", rc);
         return 1;
      }

      /* Write to disk #2 */
      cbWrite = cbActualRead;
      rc = DosWrite(hfDisk2,         /* Handle for disk 2 */
                    pBuffer,         /* Pointer to buffer */
                    cbWrite,         /* Size must be multiple of 512 */
                    cbActualWrote); /* Actual written by DosOpen */
      if (rc) {
         printf("DosWrite error return code = %u\n", rc);
         return 1;
      }
      if (cbActualRead != cbActualWrote) {
         printf("Bytes read (%u) does not equal bytes written (%u)\n",
             cbActualRead, cbActualWrote);
         return 1;
      }
      cbTotal += cbActualRead; /* Update total transferred */
   }

   printf("Transfer successfully %d bytes from disk #1 to disk #2.\n",
          cbTotal);

   /* Free allocated memmory */
   rc = DosFreeMem(pBuffer);
   if (rc != NO_ERROR) {
      printf("DosFreeMem error return code = %u\n", rc);
      return 1;
   }

   rc = DosClose(hfDisk1);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
   }

   rc = DosClose(hfDisk2);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
   }
return NO_ERROR;
}
```

DosSetFilePtrL

Purpose

DosSetFilePtrL moves the read write pointer according to the type of move specified.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosSetFilePtrL **(HFILE hFile, LONGLONG ib, ULONG method, PLONGLONG ibActual)**

Parameters

hFile HFILE)   input
The handle returned by a previous DosOpenL function.

ib LONGLONG)   input
The signed distance (offset) to move, in bytes.

method ULONG)   input
The method of moving.

Specifies a location in the file from where the *ib* to move the read/write pointer starts. The values and their meanings are described in the following list

0	FILE_BEGIN Move the pointer from the beginning of the file.
1	FILE_CURRENT Move the pointer from the current location of the read write pointer.
2	FILE_END Move the pointer from the end of the file. Use this method to determine a file s size.

ibActual PLONGLONG)   output
Address of the new pointer location.

Returns

ulrc APIRET)   returns
Return Code.

DosSetFilePtrL returns one of the following values

0	NO_ERROR
1	ERROR_INVALID_FUNCTION
6	ERROR_INVALID_HANDLE
132	ERROR_SEEK_ON_DEVICE
131	ERROR_NEGATIVE_SEEK
130	ERROR_DIRECT_ACCESS_HANDLE

Remarks

The read/write pointer in a file is a signed 64-bit number. A negative value for *ib* moves the pointer backward in the file; a positive value moves it forward. DosSetFilePtrL cannot be used to move to a negative position in the file.

DosSetFilePtrL cannot be used for a character device or pipe.

Related Functions

- DosOpenL
- DosRead
- DosSetFileSizeL
- DosWrite

Example Code

This example opens or creates and opens a file named DOSTEST.DAT , writes to it, positions the file pointer back to the beginning of the file, reads from the file, and finally closes it.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include os2.h
#include stdio.h
#include string.h

int main(void)
HFILE  hfFileHandle   = 0L;     /* Handle for file being manipulated */
ULONG  ulAction       = 0;      /* Action taken by DosOpenL */
ULONG  ulBytesRead    = 0;      /* Number of bytes read by DosRead */
ULONG  ulWrote        = 0;      /* Number of bytes written by DosWrite */
LONGLONG  ullLocal    = 0;      /* File pointer position after DosSetFilePtr */
UCHAR  uchFileName20  = "dostest.dat",     /* Name of file */
uchFileData100 = " ";              /* Data to write to file */
APIRET rc             = NO_ERROR;            /* Return code */

/* Open the file test.dat.  Use an existing file or create a new */
/* one if it doesn't exist.                                      */
rc = DosOpenL(uchFileName,                    /* File path name */
hfFileHandle,                  /* File handle */
ulAction,                      /* Action taken */
(LONGLONG)100,                  /* File primary allocation */
FILE_ARCHIVED | FILE_NORMAL,    /* File attribute */
OPEN_ACTION_CREATE_IF_NEW |
OPEN_ACTION_OPEN_IF_EXISTS,     /* Open function type */
OPEN_FLAGS_NOINHERIT |
OPEN_SHARE_DENYNONE  |
OPEN_ACCESS_READWRITE,          /* Open mode of the file */
0L);                            /* No extended attribute */if (rc != NO_ERROR)
printf("DosOpenL error return code = %u\n", rc);
return 1;
 else
printf ("DosOpenL Action taken = %ld\n", ulAction);
 /* endif */

/* Write a string to the file */
strcpy (uchFileData, "testing...\n1...\n2...\n3\n");

rc = DosWrite (hfFileHandle,                /* File handle */
(PVOID) uchFileData,         /* String to be written */
sizeof (uchFileData),        /* Size of string to be written */
ulWrote);                   /* Bytes actually written */

if (rc != NO_ERROR)
printf("DosWrite error return code = %u\n", rc);
return 1;
 else
printf ("DosWrite Bytes written = %u\n", ulWrote);
 /* endif */

/* Move the file pointer back to the beginning of the file */
rc = DosSetFilePtrL (hfFileHandle,           /* File Handle */
(LONGLONG)0,            /* Offset */
FILE_BEGIN,             /* Move from BOF */
ullLocal);             /* New location address */
if (rc != NO_ERROR)
printf("DosSetFilePtrL error return code = %u\n", rc);
return 1;


/* Read the first 100 bytes of the file */
rc = DosRead (hfFileHandle,                /* File Handle */
uchFileData,                 /* String to be read */
100L,                        /* Length of string to be read */
ulBytesRead);               /* Bytes actually read */

if (rc != NO_ERROR)
printf("DosRead error return code = %u\n", rc);
return 1;
 else
printf ("DosRead Bytes read = %u\n%s\n", ulBytesRead, uchFileData);
 /* endif */

rc = DosClose(hfFileHandle);                /* Close the file */

if (rc != NO_ERROR)
printf("DosClose error return code = %u\n", rc);
return 1;

return NO_ERROR;
```

DosSetFileSizeL

Purpose

DosSetFileSizeL changes the size of a file.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosSetFileSizeL **(HFILE hFile, LONGLONG cbSize)**

Parameters

hFile HFILE)   input
The handle of the file whose size to be changed.

cbSize LONGLONG)   input
The new size, in bytes, of the file.

Returns

ulrc APIRET)   returns
Return Code.

DosSetFileSizeL returns one of the following values

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
26	ERROR_NOT_DOS_DISK
33	ERROR_LOCK_VIOLATION
87	ERROR_INVALID_PARAMETER
112	ERROR_DISK_FULL

Remarks

When DosSetFileSizeL is issued, the file must be open in a mode that allows write access.

The size of the open file can be truncated or extended. If the file size is being extended, the file system tries to allocate additional bytes in a contiguous (or nearly contiguous) space on the medium. The values of the new bytes are undefined.

Related Functions

- DosOpenL
- DosQueryFileInfo
- DosQueryPathInfo

Example Code

This example writes to a file named DOSMAN.DAT , resets the buffer, and changes the file size.

```
#define INCL_DOSFILEMGR          /* File Manager values */
#define INCL_DOSERRORS           /* DOS Error values    */
#include os2.h
#include stdio.h
#include string.h

int main(VOID)
HFILE  hfFileHandle   = 0L;     /* Handle for file being manipulated */
ULONG  ulAction       = 0;      /* Action taken by DosOpenL */
ULONG  ulWrote        = 0;      /* Number of bytes written by DosWrite */
UCHAR  uchFileName20  = "dosman.dat",     /* Name of file */
uchFileData4   = "DATA";            /* Data to write to file */
APIRET rc             = NO_ERROR;            /* Return code */

/* Open the file dosman.dat.  Use an existing file or create a new */
/* one if it doesn't exist.                                       */
rc = DosOpenL(uchFileName, hfFileHandle, ulAction, (LONGLONG)4,
FILE_ARCHIVED | FILE_NORMAL,
OPEN_ACTION_CREATE_IF_NEW | OPEN_ACTION_OPEN_IF_EXISTS,
OPEN_FLAGS_NOINHERIT | OPEN_SHARE_DENYNONE  |
OPEN_ACCESS_READWRITE, 0L);
if (rc != NO_ERROR)
printf("DosOpenL error return code = %u\n", rc);
return 1;


rc = DosWrite (hfFileHandle, (PVOID) uchFileData,
sizeof (uchFileData), ulWrote);
if (rc != NO_ERROR)
printf("DosWrite error return code = %u\n", rc);
return 1;


rc = DosResetBuffer (hfFileHandle);
if (rc != NO_ERROR)
printf("DosResetBuffer error return code = %u\n", rc);
return 1;
 /* endif */

rc = DosSetFileSizeL (hfFileHandle, (LONGLONG)8);    /* Change file size */
if (rc != NO_ERROR)
printf("DosSetFileSizeL error return code = %u\n", rc);
return 1;


return NO_ERROR;
```

DosSetPathInfo

Purpose

DosSetPathInfo sets information for a file or directory.

Syntax

```
#define INCLDOSFILEMGR
#include os2.h
```

APIRET DosSetPathInfo **(PSZ pszPathName, ULONG ulInfoLevel, PVOID pInfoBuf, ULONG cbInfoBuf, ULONG flOptions)**

Parameters

pszPathName PSZ) input

Address of the ASCIIZ full path name of the file or subdirectory.

Global file-name characters are not permitted.

DosQuerySysInfo is called by an application during initialization to determine the maximum path length allowed by the operating system.

ulInfoLevel ULONG) input

The level of file directory information being defined.

A value of 1, 11, or 2 can be specified, as shown in the following list.

1	FIL_STANDARD Level 1 file information
11	FIL_STANDARDL Level 11 file information
2	FIL_QUERYEASIZE Level 2 file information

The structures described in *pInfoBuf* indicate the information being set for each of these levels.

pInfoBuf PVOID) input

Address of the storage area containing the file information being set.

Level 1 File Information (*ulInfoLevel* == FIL_STANDARD)
pInfoBuf contains the FILESTATUS3 data structure.

Level 11 File Information (*ulInfoLevel* == FIL_STANDARDL)
pInfo contains the FILESTATUS3L data structure, to which file information is returned.

Level 2 File Information (*ulInfoLevel* == FIL_QUERYEASIZE)
pInfoBuf contains an EAOP2 data structure.

Level 2 sets a series of extended attribute (EA) name/value pairs.

Input	*pInfoBuf* contains an EAOP2 data structure. *fpGEA2List* is ignored. *fpFEA2List* points to a data area where the relevant FEA2 list is to be found. *oError* is ignored. The FEA2 data structures must be aligned on a doubleword boundary. Each *oNextEntryOffset* field must contain the number of bytes from the beginning of the current entry to the beginning of the next entry in the FEA2 list. The *oNextEntryOffset* field in the last entry of the FEA2 list must be zero.
Output	*fpGEA2List* and *fpFEA2List* are unchanged. The area that *fpFEA2List* points to is unchanged. If an error occurred during the set, *oError* is the offset of the FEA2 entry where the error occurred. The return code is the error code corresponding to the condition that caused the error. If no error occurred, *oError* is undefined.

cbInfoBuf ULONG) input

The length, in bytes, of *pInfoBuf* .

flOptions ULONG) input

Information on how the set operation is to be performed.

If *flOptions* is 0x00000010 (DSPI_WRTTHRU), then all the information, including extended attributes (EAs), must be written to the disk before returning to the application. This guarantees that the EAs have been written to the disk. All other bits are reserved, and must be zero.

Returns

ulrc APIRET) returns

Return Code.

DosSetPathInfo returns one of the following values

0	NO_ERROR
2	ERROR_FILE_NOT_FOUND
3	ERROR_PATH_NOT_FOUND
32	ERROR_SHARING_VIOLATION
87	ERROR_INVALID_PARAMETER
124	ERROR_INVALID_LEVEL
206	ERROR_FILENAME_EXCED_RANGE
122	ERROR_INSUFFICIENT_BUFFER
254	ERROR_INVALID_EA_NAME
255	ERROR_EA_LIST_INCONSISTENT

Remarks

To use DosSetPathInfo to set any level of file information for a file or subdirectory, a process must have exclusive write access to the closed file object. Thus, if the file object is already accessed by another process, any call to DosSetPathInfo will fail.

A value of 0 in the date and time components of a field causes that field to be left unchanged. For example, if both last write date and last write time are specified as 0 in the Level 1 information structure, then both attributes of the file are left unchanged. If either last write date or last write time are other than 0, then both attributes of the file are set to the new values.

For data integrity purposes, the Write-Through bit in *flOptions* should be used only to write the extended attributes to the disk immediately, instead of caching them and writing them later. Having the Write-Through bit set constantly can degrade performance.

In the FAT file system, only the dates and times of the last write can be modified. Creation and last-access dates and times are not affected.

The last-modification date and time will be changed if the extended attributes are modified.

Related Functions

- DosEnumAttribute
- DosQueryFileInfo
- DosQueryPathInfo
- DosQuerySysInfo
- DosSetFileInfo

Example Code

This example creates a directory named HIDEME , makes it hidden, and finally deletes it.

```
#define INCL_DOSFILEMGR   /* File Manager values */
#define INCL_DOSERRORS    /* DOS Error values    */
#include os2.h
#include stdio.h
#include string.h

int main(VOID)
UCHAR       achNewDir256  = "\\HIDEME";           /* Directory name    */
FILESTATUS3 fsts3PathInfo   = 0;                /* Directory info    */
ULONG       ulBufferSize    = sizeof(FILESTATUS3);  /* Buffer size       */
APIRET      rc              = NO_ERROR;             /* Return code       */

rc = DosCreateDir(achNewDir, (PEAOP2) NULL);        /* Create directory
with no EAs       */
if (rc != NO_ERROR)
printf("DosCreateDir error return code = %u\n", rc);
return 1;
 else
printf("Directory %s created.\n",achNewDir);


rc = DosQueryPathInfo(achNewDir, FIL_STANDARD,
fsts3PathInfo, ulBufferSize); /* Get standard info */
if (rc != NO_ERROR)
printf("DosQueryPathInfo error return code = %u\n", rc);
```

```
return 1;


fsts3PathInfo.attrFile  = FILE_HIDDEN;   /* Add HIDDEN attribute to path */

rc = DosSetPathInfo(achNewDir,           /* Change directory info on     */
FIL_STANDARD,                            /* the disk using the buffer    */
fsts3PathInfo,                   /*just updated.             */
ulBufferSize,
DSPI_WRTTHRU );      /* Write data before returning  */
if (rc != NO_ERROR)
printf("DosSetPathInfo error return code = %u\n", rc);
return 1;
 else
printf("Directory %s hidden.\n",achNewDir);
/* Delete the hidden directory.  If this step is omitted, the directory
can still be manipulated by standard OS/2 commands like CHDIR and
RMDIR, it will just not be displayed in a DIR command without the
/AH display option specified.                                     */

rc = DosDeleteDir (achNewDir);
if (rc != NO_ERROR)
printf ("DosDeleteDir error  return code = %u\n", rc);
return 1;
 else
printf("Directory %s deleted.\n",achNewDir);


return NO_ERROR;
```

DosSetProcessorStatus

Purpose

DosSetProcessorStatus sets the ONLINE or OFFLINE status of a processor on an SMP system. The processor status may be queried using DosGetProcessorStatus. ONLINE status implies the processor is available for running work. OFFLINE status implies the processor is not available for running work. The processor that executes DosSetProcessorStatus must be ONLINE.

Syntax

```
#define INCL_DOS
#define INCL_DOSSPINLOCK
#include os2.h>
```

APIRET DosSetProcessorStatus **(ULONG procid, ULONG status)**

Parameters

procid (ULONG)   input
Processor ID numbered from 1 through n, where there are n processors in total.

status (ULONG)   input
Status is defined as follows

PROC_OFFLINE 0x00000000	Processor is offline.
PROC_ONLINE 0x00000001	Processor is online.

Returns

ulrc (APIRET)   returns
Return Code.

DosSetProcessorStatus returns one of the following values

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

Related Functions

- DosGetProcessorStatus

Example Code

```
int main(int argc, char *argv[], char *envp[])
{
   APIRET rc;
   ULONG procid;
   ULONG status;
   int i;

   if (argc  3) {
      printf("Syntax SETPROC  ON|OFF\n");
      return 0;
   } /* endif */

   if (strcmpi(argv[argc-1],"OFF")==0) status = 0;
   else if (strcmpi(argv[argc-1],"ON")==0) status = 1;
   else {
      printf("Syntax SETPROC  ON|OFF\n");
      return 0;
   } /* endif */

   for (i=1; iargc-1; ++i ) {
      procid = atol(argv[i]);
      rc = DosSetProcessorStatus(procid, status);
      if (rc) printf("DosSetProcesorStatus returned %u\n",rc);
   } /* endfor */

   return rc;
}
```

DosSetThreadAffinity

Purpose

DosSetThreadAffinity allows the calling thread to change the processor affinity mask for the current thread.

Syntax

APIRET DosSetThreadAffinity **(PMPAffinity pAffinityMask)**

Parameters

pAffinityMask (PMPAffinity)   input
Address of an MPAFFINITY structure that will become the current thread's affinity mask.

Returns

ulrc APIRET)   returns
Return Code.

DosSetThreadAffinity returns one of the following values

13	ERROR_INVALID_DATA
87	ERROR_INVALID_PARAMETER

Remarks

The processor affinity mask contains 1 bit per processor. A maximum of 64 processors can be designated. If affinity bits are on for non-existent processors, the error ERROR_INVALID_DATA will be returned.

Related Functions

- DosQueryThreadAffinity

Example Code

```
#define INCL_DOS
#define INCL_32
#define INCL_DOSERRORS
#define INCL_NOPMAPI
#include os2.h>
#include stdio.h>

int main(void)
{
APIRET rc;
MPAFFINITY affinity;

rc = DosSetThreadAffinity(affinity);
printf("Set thread's affinity rc = %08.8xh\n", rc);
printf("Set thread's affinity affinity[0] = %08.8xh, affinity[1] = %08.8xh\n",
        affinity.mask[0], affinity.mask[1]);
return rc;
}
```

Dos16SysTrace

Purpose

Dos16SysTrace writes a trace record to the system trace buffer. It provides a high speed event recording mechanism which may be used by PM and non-PM threads in ring 3 and ring 2 and by detached processes.

Syntax

```
#define INCL_DOSMISC
#include os2.h>
```

APIRET16 APIENTRY16 Dos16SysTrace **(USHORT major, USHORT cBuffer, USHORT minor, PCHAR pBuffer)**

Parameters

major (USHORT) input
Major code which identifies the trace record. Range reserved for user. Use is 245 255.

Valid range 0 255

cBuffer (USHORT) input
Length of optional buffer. Valid range

0 512 (before 4.0 FP 10 and 3.0 FP 35)

0 4099 (from 3.0 FP35 and 4.0 FP10 onwards).

minor (USHORT) input
Minor code which identifies the trace record. Major-minor code pair should uniquely identify the trace record.

Valid range 1 255

pBuffer (PCHAR) input
Pointer to optional buffer. If cBuffer is 0, then pBuffer is ignored.

Returns

ulrc (APIRET) returns
Return Code.

Dos16SysTrace returns one of the following values

0	NO_ERROR
150	ERROR_SYSTEM_TRACE

Remarks

Dos16SysTrace creates a trace record that includes the following items

- Header Major code, minor code, time stamp, PID of logging process
- Optional System Data Controlled by the TRACE command
- Optional User Data Specified by the pBuffer parameter

If you use Dos16SysTrace, then you need to LINK specifying OS2386.LIB. If you use DosSysTrace, then you need to LINK specifying OS2286.LIB as an additional library file with the LINK386 command.

Related Functions

- DosDumpProcess
- DosForceSystemDump

Example Code

```
int main(int argc, char *argv[], char *envp[])
{
   APIRET16 rc=0;           /* default return code */
   USHORT major=255;        /* default major code */
   USHORT minor=1;          /* default minor code */
   USHORT cBuffer=0;        /* default buffer length */
   PCHAR  pBuffer=NULL;     /* default buffer address */

   if (argc>1)
   {
      pBuffer = argv[1];
      cBuffer = strlen(argv[1]);
   }

   if (argc>2) major = atol(argv[2]);
   if (argc>3) minor = atol(argv[3]);

   rc = Dos16SysTrace(major, cBuffer, minor, pBuffer);

   if (rc) printf("DosSysTrace retuned rc=%u\n", rc);

   return rc;
}
```

DosTmrQueryFreq

Purpose

DosTmrQueryFreq queries the frequency of the high resolution timer. To get the high resolution time interval in seconds, subtract two 64 bit times and divide by the frequency.

Syntax

```
#define INCL_DOSPROFILE
#include os2.h>
```

APIRET APIENTRY DosTmrQueryFreq **(PQWORD freq)**

Parameters

freq(PQWORD)   output

Returns

ulrc (APIRET) returns
Return Code.

DosTmrQueryFreq returns one of the following values

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
99	ERROR_DEVICE_IN_USE
535	ERROR_TMR_NO_DEVICE

Example Code

```
void int3(void);

int main(int argc, char *argv[], char *envp[])
{
   QWORD start_time;
   QWORD end_time;
   QWORD interval;

   ULONG freq;
   APIRET rc;

   rc=DosTmrQueryTime(start_time);
   printf("DosTmrQueryTime rc=%u time=0x%08x%08x\n", rc,start_time.ulHi,start_time.ulLo);

   DosSleep(100);
   printf("Sleeping 100ms\n");

   rc=DosTmrQueryTime(end_time);
   printf("DosTmrQueryTime rc=%u time=0x%08x%08x\n", rc,end_time.ulHi,end_time.ulLo);

   rc=DosTmrQueryFreq(freq);
   printf("DosTmrQueryFreq rc=%u freq=%uHz\n",rc,freq);

   interval.ulLo = end_time.ulLo - start_time.ulLo;
   interval.ulHi = (end_time.ulLo >= start_time.ulLo) ?
                      end_time.ulHi - start_time.ulHi
                      end_time.ulHi - start_time.ulHi - 1;
   printf("Time interval=0x%08x%08x units=%uns\n",interval.ulHi,interval.ulLo,1000000000/freq);
   if (interval.ulHi == 0) printf("Appox. %uns\n",interval.ulLo*(1000000000/freq));

   return 0;
}
```

DosTmrQueryTime

Purpose

DosTmrQueryTime queries the 64 bit high resolution timer.

Syntax

```
#define INCL_DOSPROFILE
#include os2.h>
```

APIRET APIENTRY DosTmrQueryTime **(PQWORD time)**

Parameters

time(PQWORD) output

Returns

ulrc (APIRET)   returns

Return Code.

DosTmrQueryTime returns one of the following values

0	NO_ERROR
87	ERROR_INVALID_PARAMETER
99	ERROR_DEVICE_IN_USE
535	ERROR_TMR_NO_DEVICE
536	ERROR_TMR_INVALID_TIME

Example Code

```
void int3(void);

int main(int argc, char *argv[], char *envp[])
{
   QWORD start_time;
   QWORD end_time;
   QWORD interval;

   ULONG freq;
   APIRET rc;

   rc=DosTmrQueryTime(start_time);
   printf("DosTmrQueryTime rc=%u time=0x%08x%08x\n",rc,start_time.ulHi,start_time.ulLo);

   DosSleep(100);
   printf("Sleeping 100ms\n");

   rc=DosTmrQueryTime(end_time);
   printf("DosTmrQueryTime rc=%u time=0x%08x%08x\n",rc,end_time.ulHi,end_time.ulLo);

   rc=DosTmrQueryFreq(freq);
   printf("DosTmrQueryFreq rc=%u freq=%uHz\n",rc,freq);

   interval.ulLo = end_time.ulLo - start_time.ulLo;
   interval.ulHi = (end_time.ulLo >= start_time.ulLo) ?
                    end_time.ulHi - start_time.ulHi
                    end_time.ulHi - start_time.ulHi - 1;
   printf("Time interval=0x%08x%08x units=%uns\n",interval.ulHi,interval.ulLo,1000000000/freq);
   if (interval.ulHi == 0) printf("Appox. %uns\n",interval.ulLo*(1000000000/freq));

   return 0;
}
```

DosVerifyPidTid

Purpose

DosVerifyPidTid validates a PID/TID pair. If the thread and process exist, then a zero return code is set; otherwise the return code indicates whether the thread or the process is invalid.

Syntax

```
#define INCL_DOSMISC
#include os2.h>
```

APIRET APIENTRY DosVerifyPidTid **(PID pid, TID tid)**

Parameters

pid (PID)   input

tid (TID)   input

Returns

ulrc (APIRET)   returns
Return Code.

DosVerifyPidTid returns one of the following values

0	NO_ERROR
303	ERROR_INVALID_PROCID
309	ERROR_INVALID_THREADID

Related Functions

- DosCreateThread
- DosExecPgm

Example Code

```
int main(int argc, char *argv[], char *envp[])
{
   PID pid=0;
   TID tid=1;
   int i;
   APIRET rc;

   if (argc2) {
      printf("VPIDTID /P=pid [/T=tid]\n");
      return;
   } /* endif */

   for (i=1; i(argc ;++i ) {
   if (strnicmp (argv[i],"/P=",3)==0) pid=strtoul (argv[i]+3, NULL,16);
   else if (strnicmp (argv[i],"/T=",3)==0)
   tid=strtoul (argv[i]+3, NULL,16);
   } /* endfor */

   if (pid == 0) {
      printf("VPIDTID /P=pid [/T=tid\n");
      return;
   } /* endif */

   rc=DosVerifyPidTid (pid,tid);

   printf("Verify pid=0x%04x tid=0x%04x rc=%u\n", pid,tid,rc);

   return 0;
}
```

DosWrite

Purpose

DosWrite writes a specified number of bytes from a buffer to the specified disk

Syntax

```
#define INCL_DOSFILEMGR
#include os2.h>
```

APIRET DosWrite **(HFILE hFile, PVOID pBuffer, ULONG cbWrite, PULONG pcbActual)**

Parameters

hFile (HFILE)   input
File handle obtained from DosOpen.

pBuffer (PVOID)   input
Address of the buffer that contains the data to write.

cbWrite (ULONG)   input
The number of bytes to write. The raw file system requires that the number of bytes be a multiple of the sector size (512).

pcbActual (PULONG)   output
Address of the variable to receive the number of bytes actually written.

Returns

ulrc (APIRET)   returns
Return Code.

DosWrite returns one of the following values

0	NO_ERROR
5	ERROR_ACCESS_DENIED
6	ERROR_INVALID_HANDLE
19	ERROR_WRITE_PROTECT
26	ERROR_NOT_DOS_DISK
29	ERROR_WRITE_FAULT
33	ERROR_LOCK_VIOLATION
87	ERROR_INVALID_PARAMETER
109	ERROR_BROKEN_PIPE

Remarks

DosWrite begins writing at the current file pointer position. The file pointer is updated to where the write completed.

If there is not enough space on the disk or diskette to write all of the bytes specified by *cbWrite* , the DosWrite does not write any bytes. An error is returned and *pcbActual* is set to zero.

Using the raw file system on logical partitions requires you to lock and unlock the volume using the DosDevIOCtl Category 8, DSK_LOCKDRIVE and DSK_UNLOCKDRIVE. Writes will not succeed until the logical drive is locked.

The raw file system requires that the number of bytes written be a multiple of the sector size (512).

Related Functions

- DosOpen
- DosListIO
- DosRead
- DosSetFilePtr

Example Code

The following is NOT a complete usable program. It is simply intended   to provide an idea of how to use Raw I/O File System APIs (e.g. DosOpen, DosRead, DosWrite, DosSetFilePtr, and DosClose).

This example opens physical disk #1 for reading and physical disk #2   for writing. DosSetFilePtr is used to set the pointer to the beginning of the disks. Using DosRead and DosWrite, 10 megabytes of data is transferred from disk #1 to disk #2. Finally, DosClosed is issued to close the disk handles.

It is assumed that the size of each of the two disks is at least 10 megabytes.

```
#define INCL_DOSFILEMGR          /* Include File Manager APIs */
#define INCL_DOSMEMMGR           /* Includes Memory Management APIs */
#define INCL_DOSERRORS           /* DOS Error values */
#include os2.h>
```

```
#include stdio.h>
#include string.h>

#define SIXTY_FOUR_K 0x10000
#define ONE_MEG     0x100000
#define TEN_MEG     10*ONE_MEG

#define UNC_DISK1  "\\\\.\\Physical_Disk1"
#define UNC_DISK2  "\\\\.\\Physical_Disk2"

int main(void) {
   HFILE  hfDisk1        = 0;      /* Handle for disk #1 */
   HFILE  hfDisk2        = 0;      /* Handle for disk #2 */
   ULONG  ulAction       = 0;      /* Action taken by DosOpen */
   ULONG  cbRead         = 0;      /* Bytes to read */
   ULONG  cbActualRead   = 0;      /* Bytes read by DosRead */
   ULONG  cbWrite        = 0;      /* Bytes to write */
   ULONG  ulLocation     = 0;
   ULONG  cbActualWrote  = 0;      /* Bytes written by DosWrite */
   UCHAR  uchFileName1[20]  = UNC_DISK1, /* UNC Name of disk 1 */
          uchFileName2[20]  = UNC_DISK2; /* UNC Name of disk 2 */
   PBYTE  pBuffer        = 0;
   ULONG  cbTotal        = 0;

   APIRET rc             = NO_ERROR;            /* Return code */

   /* Open a raw file system disk #1 for reading */
   rc = DosOpen(uchFileName1,               /* File name */
                hfDisk1,                   /* File handle */
                ulAction,                  /* Action taken by DosOpen */
                0L,                         /* no file size */
                FILE_NORMAL,                /* File attribute */
                OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
                OPEN_SHARE_DENYNONE |       /* Access mode */
                OPEN_ACCESS_READONLY,
                0L);                        /* No extented attributes */
   if (rc != NO_ERROR) {
      printf("DosOpen error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Set the pointer to the begining of the disk */
   rc = DosSetFilePtr(hfDisk1,      /* Handle for disk 1 */
                      0L,           /* Offset must be multiple of 512 */
                      FILE_BEGIN,   /* Begin of the disk */
                      ulLocation); /* New pointer location */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Open a raw file system disk #2 for writing */
   rc = DosOpen(uchFileName2,               /* File name */
                hfDisk2,                   /* File handle */
                ulAction,                  /* Action taken by DosOpen */
                0L,                         /* no file size */
                FILE_NORMAL,                /* File attribute */
                OPEN_ACTION_OPEN_IF_EXISTS, /* Open existing disk */
                OPEN_SHARE_DENYNONE |       /* Access mode */
                OPEN_ACCESS_READWRITE,
                0L);                        /* No extented attributes */
   if (rc != NO_ERROR) {
      printf("DosOpen error rc = %u\n", rc);
      return(1);
   } /* endif */

   /* Set the pointer to the begining of the disk */
   rc = DosSetFilePtr(hfDisk2,      /* Handle for disk 1 */
                      0L,           /* Offset must be multiple of 512 */
                      FILE_BEGIN,   /* Begin of the disk */
                      ulLocation); /* New pointer location */
   if (rc != NO_ERROR) {
      printf("DosSetFilePtr error rc = %u\n", rc);
      return(1);
   } /* endif */


   /* Allocate 64K of memory for transfer operations */
   rc = DosAllocMem((PPVOID)pBuffer, /* Pointer to buffer */
                     SIXTY_FOUR_K,      /* Buffer size */
                     PAG_COMMIT |     /* Allocation flags */
                     PAG_READ |
                     PAG_WRITE);
   if (rc != NO_ERROR) {
      printf("DosAllocMem error rc = %u\n", rc);
      return(1);
   } /* endif */

   cbRead = SIXTY_FOUR_K;
   while (rc == NO_ERROR  cbTotal  TEN_MEG) {

      /* Read from #1 */
      rc = DosRead(hfDisk1,         /* Handle for disk 1 */
                   pBuffer,         /* Pointer to buffer */
                   cbRead,          /* Size must be multiple of 512 */
                   cbActualRead);  /* Actual read by DosOpen */
      if (rc) {
         printf("DosRead error return code = %u\n", rc);
         return 1;
      }

      /* Write to disk #2 */
      cbWrite = cbActualRead;
      rc = DosWrite(hfDisk2,         /* Handle for disk 2 */
                    pBuffer,         /* Pointer to buffer */
                    cbWrite,         /* Size must be multiple of 512 */
                    cbActualWrote); /* Actual written by DosOpen */
      if (rc) {
         printf("DosWrite error return code = %u\n", rc);
         return 1;
      }
      if (cbActualRead != cbActualWrote) {
         printf("Bytes read (%u) does not equal bytes written (%u)\n",
                cbActualRead, cbActualWrote);
         return 1;
      }
      cbTotal += cbActualRead; /* Update total transferred */
   }

   printf("Transfer successfully %d bytes from disk #1 to disk #2.\n",
          cbTotal);

   /* Free allocated memmory */
   rc = DosFreeMem(pBuffer);
   if (rc != NO_ERROR) {
      printf("DosFreeMem error return code = %u\n", rc);
      return 1;
   }

   rc = DosClose(hfDisk1);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
   }

   rc = DosClose(hfDisk2);
   if (rc != NO_ERROR) {
      printf("DosClose error return code = %u\n", rc);
      return 1;
   }
return NO_ERROR;
}
```

Raw File System APIs

This chapter contains an alphabetic list of the following data types.

- FILELOCKL
- FILEFINDBUF3L
- FILEFINDBUF4L
- FILESTATUS3L
- FILESTATUS4L

- MPAffinity

FILEFINDBUF3L

Definition

Find the file buffer data structure

Syntax

```
typedef struct FILEFINDBUF3L
ULONG       oNextEntryOffset
FDATE       fdateCreation
FTIME       ftimeCreation
FDATE       fdateLastAccess
FTIME       ftimeLastAccess
FDATE       fdateLastWrite
FTIME       ftimeLastWrite
LONGLONG    cbFile
LONGLONG    cbFileAlloc
ULONG       attrFile
UCHAR       cchName
CHAR        achNameCCHMAXPATHCOMP
 FILEFINDBUF3L

typedef FILEFINDBUF3L *PFILEFINDBUF3L;
```

Fields

fdateCreation FDATE)
: Date of file creation.

ftimeCreation FTIME)
: Time of file creation.

fdateLastAccess FDATE)
: Date of last access.

ftimeLastAccess FTIME)
: Time of last access.

fdateLastWrite FDATE)
: Date of last write.

ftimeLastWrite FTIME)
: Time of last write.

cbFile LONGLONG)
: Size of file.

cbFileAlloc LONGLONG)
: Allocated size.

attrFile ULONG)
: File attributes.

cchName UCHAR)
: Length of file name.

achName CCHMAXPATHCOMP   CHAR)
: File name including null terminator.

FILEFINDBUF4L

Definition

Level 12 (32-bit) information (used with EAs).

Syntax

```
typedef struct FILEFINDBUF4
ULONG       oNextEntryOffset
FDATE       fdateCreation
FTIME       ftimeCreation
FDATE       fdateLastAccess
FTIME       ftimeLastAccess
FDATE       fdateLastWrite
FTIME       ftimeLastWrite
LONGLONG    cbFile
LONGLONG    cbFileAlloc
ULONG       attrFile
ULONG       cbList
UCHAR       cchName
CHAR        achNameCCHMAXPATHCOMP
 FILEFINDBUF4

typedef FILEFINDBUF4L *PFILEFINDBUFL4;
```

Fields

oNextEntryOffset ULONG)
Offset of next entry.

fdateCreation FDATE)
Date of file creation.

ftimeCreation FTIME)
Time of file creation.

fdateLastAccess FDATE)
Date of last access.

ftimeLastAccess FTIME)
Time of last access.

fdateLastWrite FDATE)
Date of last write.

ftimeLastWrite FTIME)
Time of last write.

cbFile LONGLONG)
Size of file.

cbFileAlloc LONGLONG)
Allocated size.

attrFile ULONG)
File attributes.

cbList ULONG)
Size of the file s extended attributes.

The size is measured in bytes and is the size of the file s entire extended attribute set on the disk.

cchName UCHAR)
Length of file name.

achName CCHMAXPATHCOMP   CHAR)
File name including null terminator.

FILELOCKL

Definition

FILELOCKL data structure

Syntax

```
typedef struct FILELOCKL
LONGLONG       lOffset
LONGLONG       lRange
 FILELOCK

typedef FILELOCK *PFILELOCK;
```

Fields

lOffset LONGLONG)
Offset to the beginning of the lock (or unlock) range.

lRange LONGLONG)
Length, in bytes, of the lock (or unlock) range.

A value of 0 indicates that locking (or unlocking) is not required.

FILESTATUS3L

Definition

Level 11 (32-bit) (FIL_STANDARDL) information

Syntax

```
typedef struct FILESTATUS3L
FDATE       fdateCreation
FTIME       ftimeCreation
FDATE       fdateLastAccess
FTIME       ftimeLastAccess
FDATE       fdateLastWrite
FTIME       ftimeLastWrite
LONGLONG    cbFile
LONGLONG    cbFileAlloc
ULONG       attrFile
 FILESTATUS3L

typedef FILESTATUS3L *PFILESTATUS3L;
```

Fields

fdateCreation FDATE)
Date of file creation.

ftimeCreation FTIME)
Time of file creation.

fdateLastAccess FDATE)
Date of last access.

ftimeLastAccess FTIME)
Time of last access.

fdateLastWrite FDATE)
Date of last write.

ftimeLastWrite FTIME)
Time of last write.

cbFile LONGLONG)

File size (end of data).

cbFileAlloc LONGLONG)
File allocated size.

attrFile ULONG)
Attributes of the file.

FILESTATUS4L

Definition

Level 12 (32-bit) (FIL_QUERYEASIZEL) information

Syntax

```
typedef struct FILESTATUS4L
FDATE       fdateCreation
FTIME       ftimeCreation
FDATE       fdateLastAccess
FTIME       ftimeLastAccess
FDATE       fdateLastWrite
FTIME       ftimeLastWrite
LONGLONG    cbFile
LONGLONG    cbFileAlloc
ULONG       attrFile
ULONG       cbList
 FILESTATUS4L

typedef FILESTATUS4L *PFILESTATUS4L;
```

Fields

fdateCreation FDATE)
Date of file creation.

ftimeCreation FTIME)
Time of file creation.

fdateLastAccess FDATE)
Date of last access.

ftimeLastAccess FTIME)
Time of last access.

fdateLastWrite FDATE)
Date of last write.

ftimeLastWrite FTIME)
Time of last write.

cbFile LONGLONG)
File size (end of data).

cbFileAlloc LONGLONG)
File allocated size.

attrFile ULONG)
Attributes of the file.

cbList ULONG)
Length of entire EA set.

ListIOL

Definition

ListIOL data structure

Syntax

```
typedef struct ListIOL
HFILE       hFile
ULONG       CmdFlag
LONGLONG    Offset
PVOID       pBuffer
ULONG       NumBytes
ULONG       Actual
ULONG       RetCode
ULONG       Reserved
ULONG       Reserved2[3]
ULONG       Reserved3[2]
 ListIOL

typedef  ListIOL * ListIOL
```

Fields

hFile HFILE)
File handle.

CmdFlag ULONG)
Command Flag.

Offset LONGLONG)
Seek offse.t

pBuffer PVOID)
Pointer to buffer.

NumBytes ULONG)
Number of bytes to read/write.

Actual ULONG)
Actual number of bytes to read/write.

RetCode ULONG)
Operation return code.

Reserved ULONG)
(Internal.)

Reserved2[3] ULONG)
(Internal).

Reserved3[2] ULONG)
(Internal).

MPAffinity

Definition

Multi-Processor affinity mask. The mask contains 1 bit per processor and supports a maximum of 64 processors.

Syntax

```
typedef struct MPAffinity
ULONG       mask [2]
MPAFFINITY

typedef  MPAffinity *MPAffinity
```

Fields

mask ULONG)

CPUs 0 through 31 in [0] and CPUs 32 through 63 in [1].

Non-existent processors are represented as reset bits (0).

IOCtls

This chapter contains the following IOCtl commands.

Category	Function	Description
08h	69h	Logical Volume Management
80h	0Eh	Query HardDrive Geometry and Physical Parameters

Logical Volume ManagementDSK_LVMMGMT(69h)

Purpose

This IOCtl may be used with any logical volume to which a drive letter has been assigned. This function will be used by FORMAT and will also be of use to those writing disk utilities for OS/2.

Parameter Packet Format

Field	Length	C Datatype
Command Information	BYTE	UCHAR
Drive Unit	BYTE	UCHAR
Table Number	WORD	USHORT
LSN	4 BYTES	ULONG

Command Information

Command information may be

0	Identify Volume Type
1	Enable Bad Block Relocation
2	Disable Bad Block Relocation
3	Get Bad Block Information
4	Get Table Size
5	Get Relocated Sector List
6	Get Relocated Data
7	Remove Relocation Table Entry
8	Clear Relocation Table
9	Get Drive Name

The Identify Volume Type command provides a way to determine whether a volume is a Compatibility or LVM Volume.

The Enable Bad Block Relocation command enables bad block relocation on the specified volume if that volume supports it.

The Disable Bad Block Relocation command disables bad block relocation on the specified volume.

Get Bad Block Information returns the total number of bad block relocations which are currently in effect for the specified volume, as well as the number of relocation tables being used to perform bad block relocation for the specified volume. There is one bad block relocation table per physical disk partition, so, for LVM volumes employing drive linking, there may be several such tables.

Get Table Size returns the number of active entries in the specified bad block relocation table, as well as the maximum number of entries that the table can hold. The size of a bad block relocation table is dependent upon the size of the partition it is supporting. Larger partitions have larger relocation tables, while smaller partitions have smaller relocation tables.

Get Relocated Sector List returns an array of Logical Sector Numbers (LSN). Each LSN in the array is a sector whose data had to be relocated because of a problem writing to that sector. The array returned is specific to a Relocation Table. The user supplied buffer must be large enough to hold the entire array. The size of the array can be determined by using the Get Table Size command to find the number of active entries in the table, and then multiplying that value by the size of a Logical Sector Number (currently, 4 bytes).

Get Relocated Data returns the data associated with a sector that appears in a relocation table for the specified volume. The user supplied buffer must be at least 512 bytes in length, because 512 bytes are returned.

Remove Relocation Table Entry removes the specified LSN from the relocation tables on the specified volume. This function is typically used by utilities which adjust the file system on a volume so that all LSNs requiring relocation are removed from use. Since the file system will never use these LSNs again, they can be safely removed from the relocation tables for the volume, thereby freeing those entries to be used again.

Clear Relocation Table is used to remove all of the entries in a relocation table in a single operation. This function is intended to be used by FORMAT immediately before a long format is performed. Typically, FORMAT will disable bad block relocation and clear the bad block relocation tables prior to a long format so that all bad sectors may be detected by FORMAT. FORMAT will place any bad sectors detected into the bad block list for the appropriate file system.

Get Drive Name is used to return the user defined name associated with the physical drive on which the specified relocation table resides. This function can be used to identify which physical drive contains a specific relocation table associated with a volume. The name returned will not exceed 20 characters.

Drive Unit

Drive Unit is used only when the IOCtl is issued without using a previously allocated file handle. In this case, the IOCtl must be issued with a file handle of 1. Drive Unit values are 0=A, 1=B, 2=C, etc.

Table Number

Table Number is the number of the relocation table to operate on. This field is not used for commands 0 3, and 6 8.

LSN

LSN is the logical sector number of sector requiring relocation. This field is used only by commands 6 and 7.

Data Packet Format

Field	Length	C Datatype
Return Value	BYTE	UCHAR
Buffer	4 BYTEs	void*

Return Value

Return Value is set by every command that this IOCtl accepts. The specific meaning of the value it is set to is dependent upon the command issued.

Buffer

Buffer is a pointer to an area of memory large enough to hold any return value associated with the command issued. Some commands do not make use of Buffer. For these commands, Buffer should be NULL.

Values Returned

Command Information and possible return values are

Command Information	Return Value	Buffer
0	1 = Compatibility V 2 = Logical Volume	Unused
1	0 = Success 1 = Failure	Unused
2	0 = Success 1 = Failure	Unused

3	0 = Success
	1 = Failure
		typedef struct_BadBlockInf
		ULONG Total_Relocations;
		ULONG Total_Tables;
		}BadBlockInfo;

4	0 = Success
	1 = Failure
		typedef struct_BadBlackInf
		ULONG Active_Relocations;
		ULONG Max_Relocations;
		}BadBlockTableInfo;

5	0 = Success 1 = Failure	Array of LSNs. Each entry in array is sector requiring relocatio

```
6                   0 = Success         Data written to
                    1 = Failure         specified sector
7                   0 = Success         Unused
                    1 = Failure
8                   0 = Success         Unused
                    1 = Failure
9                   0 = Success         Text of name being
                    1 = Failure         returned by this function.
                                        Name will be null terminat
```

Returns

Possible values are shown in the following list

0	NO_ERROR
6	ERROR_INVALID_HANDLE
15	ERROR_INVALID_DRIVE
31	ERROR_GEN_FAILURE
87	ERROR_INVALID_PARAMETER

Query Hard Drive Geometry and Physical ParametersOEMHLP_QU (0Eh)

Purpose

This function returns geometry and physical parameters about the specified physical hard disk, if available. This information is acquired from BIOS via INT 13h function 48h at system boot.

Parameter Packet Format

```
Field                            Length
Drive Number                     BYTE
```

Drive Number
The BIOS drive number for which geometry and physical parameters are requested. The value must be 80h or greater.

Data Packet Format

```
Field                                     Length
Reserved                                  WORD
Information Flags                         WORD
Number of Physical Cylinders              DWORD
Number of Physical Heads                  DWORD
Number of Sectors Per Track               DWORD
Number of Physical Sectors                QWORD
Number of Bytes in a Sector               WORD
Reserved                                  DWORD
I/O Port Base Address                     WORD
Control Port Address                      WORD
Head Register Upper Nibble                BYTE
Reserved                                  BYTE
IRQ Information                           BYTE
Block Count for ATA R/W Multiple          BYTE
DMA Information                           BYTE
PIO Information                           BYTE
BIOS Selected Hardware Option Flags       WORD
Reserved                                  WORD
DPT Extension Revision                    BYTE
```

Information Flags
Bits are defined as follows

Bit   Description

0	DMA boundary errors are handled transparently
1	The geometry returned in bytes 4 15 is valid
2	Media is removable. Bits 4 6 are not valid if this bit is 0
3	Device supports write verify
4	Device has media change notification
5	Media is lockable
6	Device geometry is set to maximum and no media is present when this bit is set to 1.
7 15	Reserved

Number of Physical Cylinders
The number of physical cylinders on the physical drive. This field is valid only if BIT 1 of the information flags is set to 1.

Number of Physical Heads
The number of physical heads on the physical drive. This field is valid only if BIT 1 of the information flags is set to 1.

Number of Sectors Per Track
The number of sectors per track on the physical drive. This field is valid only if BIT 1 of the information flags is set to 1.

Number of Physical Sectors
The number of physical sectors on the physical drive.

Number of Bytes in a Sector
The number of bytes per sector on the physical drive.

I/O Port Base Address
This word is the address of the data register in ATA Command Block.

Control Port Address
This word is the address of the ATA Control Block Register.

Head Register Upper Nibble
The upper nibble of this byte is logically ORed with the head number, or upper 4 bits of the LBA, each time the disk is accessed.

Bit	Description
0 3	0
4	ATA DEV bit
5	1
6	LBA enabled (1 = enabled)
7	1

IRQ Information
Bits are defined as follows

Bit	Description
0 3	IRQ for this drive
4 7	0

Block Count for ATA R/W Multiple
If the hard disk was configured to use the READ/WRITE MULTIPLE command, then this field contains the block size of the transfer in sectors.

DMA Information
If the BIOS has configured the system to perform multi-word DMA transfers in place of the normal PIO transfers, this field specified the DMA mode in the upper nibble, as per the ATA-2 or later definition, and the DMA Channel in the lower nibble. ATA Channels which conform to SFF-8038i set the DMA channel to 0. Note that the DMA Type field does not follow the format of the data returned by the drive. The value of the DMA mode is not limited to 2.

Bit	Description
0 3	DMA Channel
4 7	DMA Type

PIO Information

If the BIOS has configured the system to perform PIO data transfers other than mode 0, this field specifies the PIO mode as per the ATA-2 or later definition.

Bit	Description
0 3	PIO Type
4 7	0

BIOS Selected Hardware Option Flags

Bits are defined as follows

Bit	Description	
0	Fast PIO accessing enabled	
1	DMA accessing enabled	
2	ATA READ/WRITE MULTIPLE accessing enabled	
3	CHS translation enabled	
4	LBA translation enabled	
5	Removeable media	
6	ATAPI device	
7	32 bit transfer mode	
8	ATAPI device uses command packet interrupt	
9 10	Translation type	
	00	Bit-shift translation
	01	LBA assisted translation
	10	Reserved
	11	Vendor specific translation
11	Ultra DMA accessing enabled	
12 15	Reserved	

DPT Extension Revision

Revision of DPT Extension provided by BIOS

Returns

Possible values are shown in the following list

0	NO_ERROR
87	ERROR_INVALID_PARAMETER

Remarks

Information in the data packet will be filled in, if BIOS supports INT 13h Function 48h and if BIOS can access the entire hardfile through INT 13h Function 42h and Function 43h. If either condition is not met, all fields in the data packet with be 0.

Network APIs

This chapter contains an alphabetic list of the following Server Category APIs.

- NetServerNameAdd or Net32ServerNameAdd
- NetServerNameDel or Net32ServerNameDel

- NetServerNameEnum or Net32ServerNameEnum

A number of Network APIs are multiple server name aware and will work in the context of a servername, if the first parameter (the servername) is provided. If no servername is provided and the API is issued locally, then the information returned may be for shares, device queues, or sessions that exist across all server names. If name conflicts exist, such as two shares with the same name on different servernames, the API may act on the first match it finds when no servername has been provided.The APIs that are multiple server name aware include

Serial Device Category

- NetCharDevQEnum
- NetCharDevQGetInfo
- NetCharDevQSetInfo
- NetCharDevQPurge
- NetCharDevQPurgeSelf

Share Category

- NetShareEnum
- NetShareGetInfo
- NetShareSetInfo
- NetShareAdd
- NetShareDel
- NetShareCheck

Session Category

- NetSessionEnum
- NetSessionGetInfo
- NetSessionDel

Connection Category

- NetConnectionEnum

NetServerNameAdd or Net32ServerNameAdd

Purpose

NetServerNameAdd adds a secondary server computername to a server allowing network requests directed to the secondary server name to be received and processed by the server.

Syntax

```
#include netcons.h>
#include server.h>
```

NetServerNameAdd **(const UCHAR LSFAR* pszServerName, const UCHAR LSFAR* pszAddName)**

Net32ServerNameAdd **(const UCHAR LSFAR* pszServerName, const UCHAR LSFAR* pszAddName)**

Parameters

pszServerName (const UCHAR LSFAR*) input
Points to a NULL-terminated string containing the name of the server to be added.

pszAddName (const UCHAR LSFAR*) input

Returns

ulrc (APIRET) returns for 32 bit

usrc (USHORT) returns for 16 bit
Return Code.

NetServerNameAdd or Net32ServerNameAdd returns one of the following values

0	NERR_Success
5	ERROR_ACCESS_DENIED
52	ERROR_DUP_NAME
53	ERROR_BAD_NETPATH
54	ERROR_NETWORK_BUSY
56	ERROR_TOO_MANY_CMDS
59	ERROR_UNEXP_NET_ERR
68	ERROR_TOO_MANY_NAMES
71	ERROR_REQ_NOT_ACCEP
87	ERROR_INVALID_PARAMETER
2102	NERR_NetNotStarted
2114	NERR_ServerNotStarted
2140	NERR_InternalError
2141	NERR_BadTransactConfig
2142	NERR_InvalidAPI
2468	NERR_TooManySrvNames

Remarks

The maximum number of names a server can support is defined by the manifest SV_MAX_SRV_NAMES in server.h.

The machine must also be properly configured for the additional Netbios names required as specified by the names parameter on the NETx= line of the IBMLAN.INI file.

This API can be called from OS/2 workstations. Administrative or server operator authority is required to call this API.

Related Functions

- NetServerNameDel
- NetServerNameEnum

Example Code

This example adds a servername called Server18 , then enumerates the server names in use and finally removes the Server18 servername.

```
#define PURE_32
#define INCL_DOS
#define INCL_DOSERRORS
#include os2.h>
#include stdio.h>
#include stdlib.h>
#include string.h>
#include netcons.h>
#include server.h>
int main(VOID)
{

   struct server_info_0 LSFAR * pBuffer; /* pointer to enum return info */
   ULONG  ulBufLen=4096;                 /* length in bytes of enum buffer */
   ULONG  ulLevel=0;                     /* enum return info level */
   ULONG  ulEntriesRead=0;               /* total entries read from enum */
   ULONG  ulEntriesAvail=0;              /* total entries available from enum */
   CHAR   achServer[CNLEN+1];            /* remote server name or '\0' */
   CHAR   achName[CNLEN+1];              /* server name to add and delete */
   ULONG  ulReturnCode=0;                /* return code */

   strcpy(achName,"Server18");         /* initialize servername to use */
   achServer[0] = '\0';                  /* initialize for local API call */

   ulReturnCode = Net32ServerNameAdd(achServer,achName);

   if (ulReturnCode == NO_ERROR)
   {
      if ((pBuffer = malloc(ulBufLen)) != NULL)
      {
         ulReturnCode = Net32ServerNameEnum(achServer,
                                           ulLevel,
                                           (unsigned char *)pBuffer,
                                           ulBufLen,
                                           ulEntriesRead,
                                           ulEntriesAvail);


         if (ulReturnCode == NO_ERROR || ulReturnCode == ERROR_MORE_DATA)
         {
            printf("Total entries read == %u\n",ulEntriesRead);
            printf("Total entries available == %u\n",ulEntriesAvail);
            printf("Server names are\n");

            while (ulEntriesRead) {
               printf("\t%s\n",pBuffer->sv0_name);  /* print out name */
               pBuffer++;                           /* advance to next entry */
               ulEntriesRead--;                     /* dec entries displayed */
            } /* endwhile */
         }
         else
         {
            printf("Net32ServerNameEnum() error return code = %u.\n",
                   ulReturnCode);
            Net32ServerNameDel(achServer,achName);
            return 1;
         }
      } else {
         printf("malloc() failed!\n");
         return 1;
      }

      ulReturnCode = Net32ServerNameDel(achServer,achName);

      if (ulReturnCode != NO_ERROR)
      {
         printf("Net32ServerNameDel() error return code = %u.\n",
                ulReturnCode);
         return 1;
      }
   }
   else
   {
       printf("Net32ServerNameAdd() error return code = %u.\n",
              ulReturnCode);
       return 1;
   }

   return NO_ERROR;
}
```

NetServerNameDel or Net32ServerNameDel

Purpose

NetServerNameDel removes a secondary server computername from a server and removes all shares and closes all sessions established to that name.

Syntax

```
#include netcons.h>
#include server.h>
```

NetServerNameDel **(const UCHAR LSFAR* pszServerName, const UCHAR LSFAR*, pszDelName)**

Net32ServerNameDel **(const UCHAR LSFAR* pszServerName, const UCHAR LSFAR*, pszDelName)**

Parameters

pszServerName(const UCHAR LSFAR*) input
Points to a NULL-terminated string containing the server name to be deleted.

pszDelName(const UCHAR LSFAR*) input

Returns

ulrc (APIRET) returns FOR 32 bit

usrc (USHORT) returns for 16 bit
Return Code.

NetServerNameDel or Net32ServerNameDel returns one of the following values

0	NERR_Success
5	ERROR_ACCESS_DENIED
53	ERROR_BAD_NETPATH
54	ERROR_NETWORK_BUSY
56	ERROR_TOO_MANY_CMDS
59	ERROR_UNEXP_NET_ERR
71	ERROR_REQ_NOT_ACCEP
87	ERROR_INVALID_PARAMETER
2102	NERR_NetNotStarted
2114	NERR_ServerNotStarted
2140	NERR_InternalError
2141	NERR_BadTransactConfig
2142	NERR_InvalidAPI
2460	NERR_NoSuchServer
2469	NERR_DelPrimaryName

Remarks

Only secondary server names can be deleted by the API. An attempt to delete the primary server name will result in NERR_DelPrimaryName being returned.

If a name successfully deleted had sessions to it, then the name may still show in NetServerNameEnum and may not be re-added until the server has completed closing all sessions established to that name.

Any shares that were added to the deleted name will also be removed.

This API can be called from OS/2 workstations. Administrative or server operator authority is required to call this API.

Related Functions

- NetServerNameAdd

- NetServerNameEnum

Example Code

This example adds a servername called Server18 , then enumerates the server names in use and finally removes the Server18 servername.

```
#define PURE_32
#define INCL_DOS
#define INCL_DOSERRORS
#include os2.h>
#include stdio.h>
#include stdlib.h>
#include string.h>
#include netcons.h>
#include server.h>
int main(VOID)
{

   struct server_info_0 LSFAR * pBuffer; /* pointer to enum return info */
   ULONG  ulBufLen=4096;                 /* length in bytes of enum buffer */
   ULONG  ulLevel=0;                     /* enum return info level */
   ULONG  ulEntriesRead=0;               /* total entries read from enum */
   ULONG  ulEntriesAvail=0;              /* total entries available from enum */
   CHAR   achServer[CNLEN+1];            /* remote server name or '\0' */
   CHAR   achName[CNLEN+1];              /* server name to add and delete */
   ULONG  ulReturnCode=0;                /* return code */

   strcpy(achName,"Server18");         /* initialize servername to use */
   achServer[0] = '\0';                  /* initialize for local API call */

   ulReturnCode = Net32ServerNameAdd(achServer,achName);

   if (ulReturnCode == NO_ERROR)
   {
      if ((pBuffer = malloc(ulBufLen)) != NULL)
      {
         ulReturnCode = Net32ServerNameEnum(achServer,
                                            ulLevel,
                                            (unsigned char *)pBuffer,
                                            ulBufLen,
                                            ulEntriesRead,
                                            ulEntriesAvail);


         if (ulReturnCode == NO_ERROR || ulReturnCode == ERROR_MORE_DATA)
         {
            printf("Total entries read == %u\n",ulEntriesRead);
            printf("Total entries available == %u\n",ulEntriesAvail);
            printf("Server names are\n");

            while (ulEntriesRead) {
               printf("\t%s\n",pBuffer->sv0_name);  /* print out name */
               pBuffer++;                           /* advance to next entry */
               ulEntriesRead--;                     /* dec entries displayed */
            } /* endwhile */
         }
         else
         {
            printf("Net32ServerNameEnum() error return code = %u.\n",
                   ulReturnCode);
            Net32ServerNameDel(achServer,achName);
            return 1;
         }
      } else {
         printf("malloc() failed!\n");
         return 1;
      }

      ulReturnCode = Net32ServerNameDel(achServer,achName);

      if (ulReturnCode != NO_ERROR)
      {
         printf("Net32ServerNameDel() error return code = %u.\n",
                ulReturnCode);
         return 1;
      }
   }
   else
   {
       printf("Net32ServerNameAdd() error return code = %u.\n",
              ulReturnCode);
       return 1;
   }

   return NO_ERROR;
}
```

NetServerNameEnum or Net32ServerNameEnum

Purpose

NetServerNameEnum enumerates the set of computernames by which a server is known by on the network.

Syntax

```
#include netcons.h>
#include server.h>
```

NetServerNameEnum **(const UCHAR LSFAR* pszServerName, SHORT sLevel, UCHAR LSFAR* buf, USHORT usBuflen, USHORT LSFAR* pusEntriesReturned, USHORT LSFAR* pusEntriesAvail)**

Net32ServerNameEnum **(const UCHAR pszServerName, ULONG ulLevel, UCHAR* Buf, ULONG ulBuflen, ULONG* pulEntriesReturned, ULONG* pulEntriesAvail)**

Parameters

pszServerName(const UCHAR LSFAR*)   input
Points to a string containing the network name of the server.

sLevel(SHORT) input
Specifies the level of detail (MBZ) for the server_info data structure, as described in Server Level 0. Other levels such as 1, 2, 3 and 20 are not valid for NetServerNameEnum.

buf(UCHAR*) output
Points to the local buffer address of the data structure to be sent or received.

usBuflen(USHORT) or (ULONG) input
Specifies the amount of local memory allocated to the buf data structure.

pusEntriesReturned(USHORT LSFAR*) or pulEntriesReturned(ULONG*) or (PULONG) output
Points to the number of data structures returned.

pusEntriesAvail(USHORT LSFAR*) orpusEntriesAvail (ULONG*) or (PULONG) output
Points to the number of data structures currently available.

ulLevel(ULONG) input
Specifies the level of detail (MBZ) for the server_info data structure, as described in Server Level 0. Other levels such as 1, 2, 3 and 20 are not valid for NetServerNameEnum.

Returns

ulrc (APIRET)   returns for 32 bit

usrc (USHORT)   returns for 16 bit
Return Code.

NetServerNameEnum or Net32ServerNameEnum returns one of the following values

0	NERR_Success
5	ERROR_ACCESS_DENIED
124	ERROR_INVALID_LEVEL
234	ERROR_MORE_DATA
2102	NERR_NetNotStarted

2114	NERR_ServerNotStarted
2140	NERR_InternalError
2141	NERR_BadTransactConfig
2142	NERR_InvalidAPI

Remarks

If you call this API with the buffer length parameter equal to zero, the API returns a value for total entries available. This technique is useful if you do not know the exact buffer size required.

The NetServerNameEnum API can obtain only level 0 data structures.

This API returns the list of server names being used by a server. This may include names in the process of being added or deleted, not just active server names.

The set of server names returned will always list the primary server name first, and if there are no other server names in use, the primary server name will be the only name returned in the return buffer.

This API can be called from OS/2 workstations. Administrative or server operator authority is required to call this API.

Related Functions

- NetServerNameAdd
- NetServerNameDel

Example Code

This example adds a servername called Server18 , then enumerates the server names in use and finally removes the Server18 " servername.

```
#define PURE_32
#define INCL_DOS
#define INCL_DOSERRORS
#include os2.h>
#include stdio.h>
#include stdlib.h>
#include string.h>
#include netcons.h>
#include server.h>
int main(VOID)
{

   struct server_info_0 LSFAR * pBuffer; /* pointer to enum return info */
   ULONG  ulBufLen=4096;                 /* length in bytes of enum buffer */
   ULONG  ulLevel=0;                     /* enum return info level */
   ULONG  ulEntriesRead=0;               /* total entries read from enum */
   ULONG  ulEntriesAvail=0;              /* total entries available from enum */
   CHAR   achServer[CNLEN+1];            /* remote server name or '\0' */
   CHAR   achName[CNLEN+1];              /* server name to add and delete */
   ULONG  ulReturnCode=0;                /* return code */

   strcpy(achName,"Server18");         /* initialize servername to use */
   achServer[0] = '\0';                  /* initialize for local API call */

   ulReturnCode = Net32ServerNameAdd(achServer,achName);

   if (ulReturnCode == NO_ERROR)
   {
      if ((pBuffer = malloc(ulBufLen)) != NULL)
      {
         ulReturnCode = Net32ServerNameEnum(achServer,
                                            ulLevel,
                                            (unsigned char *)pBuffer,
                                            ulBufLen,
                                            ulEntriesRead,
                                            ulEntriesAvail);


         if (ulReturnCode == NO_ERROR || ulReturnCode == ERROR_MORE_DATA)
         {
            printf("Total entries read == %u\n",ulEntriesRead);
            printf("Total entries available == %u\n",ulEntriesAvail);
            printf("Server names are\n");

            while (ulEntriesRead) {
               printf("\t%s\n",pBuffer->sv0_name);  /* print out name */
               pBuffer++;                           /* advance to next entry */
               ulEntriesRead--;                     /* dec entries displayed */
            } /* endwhile */
         }
         else
         {
            printf("Net32ServerNameEnum() error return code = %u.\n",
                   ulReturnCode);
            Net32ServerNameDel(achServer,achName);
            return 1;
         }
      } else {
         printf("malloc() failed!\n");
         return 1;
      }

      ulReturnCode = Net32ServerNameDel(achServer,achName);

      if (ulReturnCode != NO_ERROR)
      {
         printf("Net32ServerNameDel() error return code = %u.\n",
                ulReturnCode);
         return 1;
      }
   }
   else
   {
       printf("Net32ServerNameAdd() error return code = %u.\n",
              ulReturnCode);
       return 1;
   }

   return NO_ERROR;
}
```

Windows Functions

This chapter contains an alphabetic list of the following Windows functions

- WinHAPPfromPID
- WinHSWITCHfromHAPP
- WinRestartWorkplace
- WinWaitForShell()

WinHAPPfromPID

Purpose

WinHAPPfromPID returns the PM Application Handle (HAPP) from the process ID. If the proces ID is not a valid PM application, then 0 is returned.

Syntax

```
#define INCL_PMAPI
#include os2.h>
```

HAPP WinHAPPfromPID **(PID pid)**

Parameters

pid(PID)   input

Process ID

Returns

happp(HAPP) returns

Application handle.

NULLHANDLE	If the PID is invalid, or an error occurred.
HAPP	The Application Handle, if PID is valid.

Remarks

WinHAPPfromHSWITCH and WinHSWITCHfromHAPP may be called from non-PM programs. For some versions of OS/2 it may be necessary to import explicitly these APIs using the following ordinals

WinHAPPfromPID PMMERGE.5198

WinHSWITCHfromHAPP PMMERGE.5199

Related Functions

- WinHSWITCHfromHAPP
- WinQuerySwitchEntry
- WinQuerySwitchHandle
- WinQuerySwitchList

Example Code

```
int main (int argc, char *argv[], char *envp[])
{

   APIRET rc;
   HAPP happ;
   HSWITCH hswitch;
   SWCNTRL swcntrl;
   PID pid;

   if (argc==1) {
     printf("QSWLIST pid\n");
     return 0;
    }   /* endif */

   pid=strtoul(argv[1],NULL,0);

   happ=WinHAPPfromPID(pid);            /* get HAPP from PID */

   hswitch=WinHSWITCHfromHAPP(happ);     /* get HSWITCH from HAPP */

   rc=WinQuerySwitchEntry(hswitch, swcntrl); /* interpret HSWITCH */
   if (rc) {
     printf("WinQuerySwitchEntry returned %u\n",rc);
     return rc;
   } /* endif */

   printf("Pid %04x, Happ %08x, Hswitch %08x\n", pid, happ, hswitch);
   printf("swcntrl.hwnd     \t%08x,   swcntrl.hwndIcon    \t%08x\n",
           swcntrl.hwnd, swcntrl.hwndIcon);
   printf("swcntrl.hprog    \t%08x,   swcntrl.idProcess   \t%08x\n",
           swcntrl.hprog, swcntrl.idProcess);
   printf("swcntrl.idSession\t%08x,   swcntrl.uchVisbility\t%08x\n",
           swcntrl.idSession, swcntrl.uchVisibility);
   printf("swcntrl.fbJump   \t%08x,   swcntrl.bProgType   \t%08x\n",
           swcntrl.fbJump, swcntrl.bProgType);
   printf("swcntrl.szSwtitle %s\n", sz.Swtitle);

   return 0;
}
```

WinHSWITCHfromHAPP

Purpose

WinHSWITCHfromHAPP returns the handle of the switch list entry from the application handle. If the application handle is invalid or IF no switch list eNtry exists, then 0 is returned.

Syntax

```
#define INCL_PMAPI
#include os2.h>
```

HSWITCH WinHSWITCHfromHAPP **(HAPP happ)**

Parameters

happ(HAPP)   input
Application handle.

Returns

hswitch (HSWITCH)   returns
Switch list handle.

NULLHANDLE	If the HAPP is invalid, the Switch List Entry is not defined for this HAPP, or an error occurred.
HSWITCH	The Handle of the Switch List Entry, if HAPP is valid and a Switch List Entry exists.

Remarks

WinHAPPfromHSWITCH and WinHSWITCHfromHAPP may be called from non-PM programs. For some versions of OS/2 it may be necessary to import explicitly these APIs using the following ordinals

WinHAPPfromPID	PMMERGE.5198
WinHSWITCHfromHAPP	PMMERGE.5199

Related Functions

- WinHAPPfromPID
- WinQuerySwitchEntry
- WinQuerySwitchHandle
- WinQuerySwitchList

Example Code

```
int main (int argc, char *argv[], char *envp[])
{

   APIRET rc;
   HAPP happ;
   HSWITCH hswitch;
   SWCNTRL swcntrl;
   PID pid;

   if (argc==1) {
     printf("QSWLIST pid\n");
     return 0;    }                     /* endif */

   pid=strtoul(argv[1],NULL,0);

   happ=WinHAPPfromPID(pid);            /* get HAPP from PID */

   hswitch=WinHSWITCHfromHAPP(happ);     /* get HSWITCH from HAPP */

   rc=WinQuerySwitchEntry(hswitch, swcntrl); /* interpret HSWITCH */
   if (rc) {
     printf("WinQuerySwitchEntry returned %u\n",rc);
     return rc;
   } /* endif */

   printf("Pid %04x, Happ %08x, Hswitch %08x\n", pid, happ, hswitch);
   printf("swcntrl.hwnd     \t%08x,   swcntrl.hwndIcon    \t%08x\n",
           swcntrl.hwnd, swcntrl.hwndIcon);
   printf("swcntrl.hprog    \t%08x,   swcntrl.idProcess   \t%08x\n",
           swcntrl.hprog, swcntrl.idProcess);
   printf("swcntrl.idSession\t%08x,   swcntrl.uchVisbility\t%08x\n",
           swcntrl.idSession, swcntrl.uchVisibility);
   printf("swcntrl.fbJump   \t%08x,   swcntrl.bProgType   \t%08x\n",
           swcntrl.fbJump, swcntrl.bProgType);
   printf("swcntrl.szSwtitle %s\n", sz.Swtitle);

   return 0;
}
```

WinRestartWorkplace

Purpose

This function causes the Workplace(TM) process to terminate and re-initialize.

This function is applicable to OS/2 Warp 4, or higher, and WorkSpace On-Demand client operating systems.

Syntax

```
#include os2.h>

BOOL32 APIENTRY WinRestartWorkplace(VOID);
```

Parameters

None.

Returns

rc (BOOL32)   returns
Always returns FALSE.

Remarks

This function will cause the Workplace process to terminate and re-initialize. This call is useful in debugging workplace objects or for install programs that reregister system classes.

Example Code

This example terminates and re-initializes the Workplace process.

```
#include os2.h>
BOOL32 EXPENTRY WinRestartWorkplace(VOID);

/* Terminate and re-initialize the Workplace process */
WinRestartWorkplace();
```

WinWaitForShell()

Purpose

WinWaitForShell() determines if various events in the Workplace Shell(R) have taken place.

Syntax

```
#define
#include os2.h>
```

BOOL EXPENTRY WinWaitForShell() **(ULONG ulEvent)**

Parameters

ulEvent (ULONG)   input

ulEvent has the following flags which indicate which event is to be queried. One, and only one, of these flags must be turned on.

WWFS_DESKTOPCREATED	Desktop has been created.
WWFS_DESKTOPOPENED	A view of the Desktop has been opened.
WWFS_DESKTOPPOPULATED	The desktop has been populated..
WWFS_QUERY	Query if the event has taken place. If this flag is not turned on then this call will block until the event has taken place.

Returns

rc (BOOL)   returns

Success indicator.

True	Event has taken place
False	Event has not taken place

Possible returns from WinGetLastError

PMERR_INVALID_PARAMETER (0x1208) One of the defined WWFS_DESKTOP* flags was not passed in ulEvent.

Remarks

This function can be used to either determine if a Workpalce Shell event has taken place or wait until that event has taken place. Set ulEvent to one of the WWFS_DESKTOP* #defines above.

To block until the event has occurred, do not turn on the WWFS_QUERY flag.

Simply to query if the event has occurred and not to wait for it to occur, turn on the WWFS_QUERY flag.

Example Code

This example checks (non-blocking) to see if the Workplalce Shell Desktop to be populated has been populated or not.

```
#define
#include os2.h>

BOOL fOccurred;

fOccurred + WinWaitForShell(WWFS_DESKTOPPOPULATED | WWFS_QUERY);
if (fOccurred)
   {
    /* The Desktop has been populated */
   }
else
   {
    /* The Desktop has not been populated */
```

APIs Supporting High Memory Objects

APIs in Warp Server 4 Advanced/SMP

- DosFindFirst
- DosFindNext
- DosFreeMem
- DosGetNamedSharedMem
- DosGiveSharedMem
- DosLoadModule
- DosOpen
- DosQueryMem
- DosQueryModuleName
- DosQueryPageUsage
- DosQueryPathInfo
- DosRead
- DosSetMem
- DosSetPathInfo
- DosSubAllocMem
- DosSubFreeMem
- DosSubSetMem
- DosSubUnsetMem
- DosWrite

Additional APIs Supported in Warp Server for e-business

- DosAcknowledgeSignalException
- DosAddMuxWaitSem
- DosCloseEventSem
- DosCloseMutexSem
- DosCloseMuxWaitSem
- DosCopy
- DosCreateDir
- DosCreateEventSem
- DosCreateMutexSem
- DosCreateMuxWaitSem
- DosCreateNPipe

- DosCreateQueue
- DosCreateThread2
- DosDelete
- DosDeleteDir
- DosDeleteMuxWaitSem
- DosEnterMustComplete
- DosExecPgm
- DosExitMustComplete
- DosMove
- DosOpenEventSem
- DosOpenMutexSem
- DosOpenMutexWaitSem
- DosPhysicalDisk
- DosPostEventSem
- DosQueryCurrentDir
- DosQueryCurrentDisk
- DosQueryEventSem
- DosQueryFSAttach
- DosQueryFSInfo
- DosQueryFileInfo
- DosQueryModuleHandle
- DosQueryMutexSem
- DosQueryMuxWaitSem
- DosQueryNPHState
- DosRaiseException
- DosReadQueue
- DosReleaseMutexSem
- DosRequestMutexSem
- DosResetEventSem
- DosScanEnv
- DosSearchPath
- DosSendSignalException
- DosSetCurrentDir
- DosSetExceptionHandler
- DosSetFileInfo
- DosSetFileLocks
- DosSetRelMaxFH
- DosSetSignalExceptionFocus

- DosUnsetExceptionHandler
- DosUnwindException
- DosWaitEventSem
- DosWaitMuxWaitSem

Notices

Trademarks
